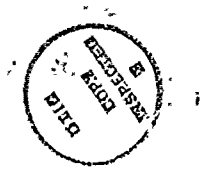
DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCE/ENG/91D-03



Logic Programming in Digital Circuit Design

THESIS

Joseph William Eicher
Captain, USAF

AFIT/GCE/ENG/91D-03

AFIT/GCE/ENG/91D-03

Logic Programming in Digital Circuit Design

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Joseph William Eicher, B.S.

Captain, USAF

December, 1991

Acknowledgments

I would like to thank my wife, Bonny, and my children, Gillian and Joshua, for their understanding and patience during the last eighteen months. As this project consumed the bulk of my time, their continued support made the effort seem more worthwhile. I would aiso like to thank Doctor Frank Brown, my thesis advisor. His support scaled insurmountable obstacles down to a size I could comfortably confront. Finally I would like to express appreciation to the members of my committee, Doctors Henry Potoczny and Gregg Gunsch, for the excellent guidance they provided me during my journey.

Joseph William Eicher

Table of Contents

List of Figures

List of Tables

AFIT/GCE/ENG/91D-03

Abstract

The design of large, complex digital circuitry requires highly skilled engineers. Much of the time spent by these engineers in the design phase involves tasks that are repetitive, tedious, and slow. If these repetitive tasks are automated, the engineer can spend more time managing the design process and produce a better-quality design in less time. Logic programming can be used to automate design tasks, even those that require a high degree of skill. This thesis investigates several aspects of the digital circuit design process that involve pattern-matching paradigms suitable for encoding in the logic programming language Prolog.

Logic Programming in Digital Circuit Design

I. Introduction

1.1 Background

Several Computer Aided Design (CAD) systems are used at the Air Force Institute of Technology (AFIT) to design electronic circuits. These CAD systems function with conventional software and work much like a hammer or screwdriver in that the quality of the final product is a direct function of the skill of the craftsman. Computers run with conventional CAD software lack the inference ability associated with computers which use artificial intelligence (AI) languages. When pattern-matching AI languages are used instead of conventional CAD software, computers can draw "intelligent" conclusions concerning electronic circuit structure from simple facts. Computers using AI software can be useful in discovering design mistakes before they become a problem [46].

AFIT currently supports research which relies heavily on conventional CAD systems. As technology progresses and electronic circuit design becomes increasingly complex, the opportunity for making a design error increases significantly. Mistakes in circuit design are always costly to correct unless they are discovered in an early phase of the design process. Design errors can be found and corrected before they become a serious problem if CAD systems are used which incorporate logic programming and knowledge-based reasoning.

The special character of logic programming with an AI language such as Prolog can be exploited to automatically design electronic circuits through knowledge-based reasoning [14:59]. Employing knowledge-based reasoning in the circuit design process can be useful in ensuring the functional correctness of a particular specification, especially when the design involves tedious tasks that are prone to human error [5:100].

1.2 Statement Of The Problem

Logic programming can be used in the design process during any phase which is dependent on pattern-matching. Circuit modelling, simulation, and transformation are some of the processes whose outcomes depend on the ability of the design engineer to detect recurring patterns. The pattern-matching ability of Prolog can be used to automate these

design processes, producing quicker, more accurate, and more consistent results. Prolog's built-in depth-first search strategy guarantees that all possible solutions to a problem will be found. This thoroughness can be used during many phases of the engineering process to ensure that the best solution to a given problem is found. This thesis investigates applications of logic programming in digital circuit modelling, simulation, and transformation.

1.3 Assumptions

The reader should have a basic understanding of logic programming with Prolog. If that is not the case, a good source for reference would be either *Programming in Prolog* [16] by W.F.Clocksin and C. Mellish or *Prolog Programming for Artificial Intelligence* [4] by Ivan Bratko. Several outstanding articles on the subject are also available for reference [1, 2, 3, 15, 17, 18, 19, 20, 24, 28, 35, 37, 38, 40, 47]. Chapter 2 contains an introduction to some of the more important aspects of logic programming which are central to this thesis. Chapter 2 is not intended to be a tutorial or extensive, all-encompassing discussion, but rather a beacon for the less informed reader to follow.

1.4 Scope

This thesis will focus on several issues associated with digital circuit design and ways in which logic programming can be exploited to automate circuit-design processes. A brief outline of logic programming with Prolog is presented. Aspects of Prolog which are central to the thesis are explained. Logic programming in general and the pattern-matching ability of Prolog in particular are used to develop algorithms which can be used to assist digital circuit design engineers by automating selected segments of the design process. Digital circuit modelling, simulation, and some aspects of circuit transformation are discussed.

1.5 Approach/Methodology

After a brief explanation of some of the features of Prolog used in this thesis, methods of digital circuit modelling with Prolog are investigated. Following the development of a modelling protocol, a simple transformation process, *extraction*, is presented. Extraction uses the pattern-matching capabilities of Prolog to detect recurring patterns in digital circuits. Several examples are shown. Circuit simulation is also discussed. Several important traits of Prolog make this language an ideal one for simulating digital circuit operation. Finally, another transformation process, called *specialization*, is developed. Specialization

can be used to eliminate one class of redundant components from a circuit, resulting in a more economical design.

1.6 Overview

The present chapter introduces applications of logic programming to digital circuit design. The concepts of modelling, simulation, and transformation are mentioned and the objectives of this research along with an overall view of assumptions and methodologies are stated.

Chapter 2 presents a brief history of logic programming together with a superficial introduction to Prolog. The main focus of this chapter is on Prolog and aspects of the language which can be exploited to aid in the design of digital circuitry. Chapter 2 is based on an extensive review of current as well as past literature on the subject.

Chapter 3 addresses the use of Prolog in modelling digital circuits. It shows how different paradigms are used in digital circuit modelling. Several examples illustrate the different methodolcgies.

Chapter 4 discusses in detail one aspect of circuit transformation, *extraction*. A simple extraction algorithm is developed and applied to a circuit composed of NAND-gates. The circuit is tested and all full-adder combinations are extracted.

Chapter 5 discusses the application of Prolog to the simulation of digital combinational and sequential circuits. Algorithms, along with sample simulations, are presented. A digital circuit simulator that operates with a subset of the 7400 series of TTL integrated circuits is presented at the end of the chapter. The simulator is easy to use and operates much faster than simulators previously developed at AFIT.

Chapter 6 highlights another transformation problem, that of attempting to achieve a more efficient design by identifying and eliminating unused components. The process, called *specialization*, eliminates all components that have unused outputs from a circuit and automatically rewrites the modified circuit and its connections.

The final chapter, Chapter 7, discusses the strengths and weaknesses of the algorithms presented and makes recommendations for the direction of future research.

Throughout this thesis, the following convention for describing logic components will be adopted:

II. An Overview of Logic Programming

2.1 Some Important Developments in the History of Automated Reasoning

2.1.1 Early Developments. Automated reasoning is based on the theory of predicate calculus. Predicate calculus was invented in 1879 by Gottlob Frége, a mathematician [42:107]. Frége's goal was to develop a system that could be used to analyze the formal structure of pure thought. He called his system *Begriffsschrift* which loosely translates to "notation for concepts". Frége envisioned his system as a universal language that could be used to represent every form of rational thought. The represented thoughts could be manipulated in a precise and mathematical way in order to deductively reason about them. Frége accepted that his system was complete. Formal proof of the completeness of Frége's system of predicate calculus did not come, however, until 1929 and 1930 when Thoralf Skölem, Kurt Gödel and Jacques Herbrand proved independently that Frége's system was a complete system of notation [42:108]. Herbrand had several versions of the proof procedure; one version used the process of *unification* which later played a major role in the development of logic programming.

Another important theoretical contribution to automated reasoning came in 1936 when A.M. Turing showed that it was possible to invent a machine that could be used to compute any sequence of numbers that were calculable by finite means. He also showed that his "automatic machine" could find all provable formulae of the predicate calculus provided the notation was systematic and involved a finite number of symbols [48:252].

The role of logic in the early development of the field of artificial intelligence was limited to meta-level reasoning about program correctness and attempts to automate theorem proving. In 1955, Evert Beth made the first attempt at programming predicate calculus proofs on a computer. The process was computationally intensive and only small proofs could be done [42:108]. A major breakthrough came in 1965 when J.A. Robinson published his *resolution principle.*

Resolution. Resolution was a major step forward in the field of automated theorem proving as well as logic programming and is one of the most powerful theorem proving techniques currently in use. It uses Herbrand's process of unification, a generalized form of pattern matching [11:70-97]. The resolution process creates the least specialized or most general common expression from two atomic relations by substituting for like variables in both of them [24:2]. The resolution process is applied to problem statements written in clause form. A logic clause is an expression of the form

$$\overbrace{B_1, \ldots, B_m}^{head} \leftarrow \underbrace{A_1, \ldots, A_n}_{body} \qquad (m, n \geq 0)$$

where atoms $B_1, \ldots, B_m$ constitute the head and represent zero or more conclusions. The body is made up of goals $A_1, \ldots, A_n$ and represent zero or more conditions [44:309]. If the clause contains the variables $x_1, \ldots, x_n$ it can be read as

for all $x_1, \ldots, x_k$
if A_1 and...and A_n then
B_1 or...or B_m.

If $n = 0$, then the clause is interpreted as

for all $x_1, \ldots, x_k$
B_1 or...or B_m.

If $m = 0$ then the clause is interpreted as

for no $x_1, \ldots, x_k$
A_1 and...and A_n.

If $m = n = 0$ then the clause is interpreted always to be false [36:426]. The basic principles of the resolution process can be illustrated in a simple way within the confines of propositional logic. Consider the two expressions

clause 1. L1 $\vee$ $L2 \vee \sim L3$
clause 2. L4 $\vee$ $L5 \vee L3$

where the Ls are arbitrary literals and $\sim$ represents the negation of a literal. Clauses are expressed as the disjunction ($\vee$) of literals. The literals are expressed as conjunctions of indivisible atoms. An atom is opposed if it appears directly in one clause and negated in another [8:7]. In the two expressions given above, L3 is opposed. The resolvent of the two expressions can be generated as follows:

1. Form an expression that contains all literals of the two parent expressions.
2. Delete the opposed pair of literals.
3. Delete any repeated literals.

Thus the resolvent of the example expression is

$$L1 \vee L2 \vee L4 \vee L5.$$

The resolvent expression generated from the two parent expressions are a logical consequence of the parent expressions. A given set of expressions will produce nothing but

legally-inferred logical consequences when subjected to the resolution process. The automation of this process resulted in the development of the logic programming language Prolog.

2.1.2 The Emergence of Prolog. The origin of Prolog can be viewed as the confluence of two different endeavors: Alain Colmerauer's interest in natural language processing, and Robert Kowalski's research into logic and theorem proving [18:26]. Their independent efforts resulted in the development of the first efficient programming language based on Frége's system of logic [47:96]. Prolog, short for *Programmation en Logique* [37:38] is based on a restricted form of automated reasoning in which logic clauses have no more than one conclusion associated with a set of conditions [42:111]. Prolog programming involves expressing information in Horn-clause form (discussed in section 2.2.1) and, using Robinson's resolution process, forming new deductions by reasoning about the logical content of the information. A Prolog interpreter can be thought of as a resolution based theorem-prover. Clocksin and Mellish [16] describe programming in Prolog as:

1. Declaring some `facts` about objects and their relationships;

2. Defining `rules` about objects and their relationships, and

3. Asking `questions` (queries) about objects and their relationships.

2.2 Prolog: A Logic Programming Language

2.2.1 Horn-Clauses. Logic Programming is a restricted form of automated reasoning based on the use of Horn-clauses. Horn-clauses are logic clauses in which no more than one conclusion is associated with a set of conditions. Horn-clause structure can be expressed as any of the forms found in Table 2.1.

Table 2.1. Horn-clause forms

Horn-Clause Form	Prolog Function
$B_1 \longleftarrow$	`fact`
$B_1 \longleftarrow A_1, \ldots, A_n$	`rule`
$\longleftarrow A_1, \ldots, A_n$	`query`

Horn-clause syntax allows the conjunction of zero or more conditions (also known as antecedents) in the clause's body to imply at most one conclusion (also known as a

consequent) in the clause's head [7]. A Horn-clause states that for all the possible values of the variables appearing in the clause, if all the conditions of the conclusions hold, then the conclusion is true. Every variable is understood to be universally quantified over the clause in which it appears; hence each clause stands alone. Horn-clauses can be thought of as procedural definitions. The head of the clause, accompanied with corresponding arguments (explained in more detail later), represents the procedure's name. The conditions of the clause, accompanied with their corresponding arguments, represent the procedure's body. All clauses with the same head/argument structure are collectively called a *procedure*. As shown in Table 2.1, Prolog clauses occur in three forms: `facts`, `rules`, and `queries`.

Facts. A `fact` is the most basic statement in Prolog. Facts are conclusions or consequents that have no conditions or antecedents; facts thus form simple statements about objects and their relationships. A fact announces that some relation is true. A finite set of facts constitutes a simple logic program. Some examples of facts are shown in Table 2.2.

Table 2.2. Examples of Prolog facts

Prolog Fact	Meaning
`likes(joe,apples).`	"joe likes apples."
`friend(bonny,joe).`	"bonny is a friend of joe."
`mother(sally,sue).`	"sally is the mother of sue."

Rules. A single program-clause can either be a fact or a `rule`. Rules define complicated relationships among objects by announcing that the head of the rule is true if all of the goals in the `body` of the rule are true. A rule has the form

$$\overbrace{B_1}^{head} \; :- \; \underbrace{A_1, \ldots, A_n}_{body} \qquad (n \geq 0)$$

which is the same form as that given earlier in Table 2.1 for Horn-clause rules. Note that a fact is a special form of a rule when n = 0. The :- symbol is used in Prolog programming to replace the ← and can be read as the word "if" or "is implied by" [34:145]. Some examples of rules are

Rule	Meaning

`happy(X) :-`
`  has(X,Y),`
`  dog(Y).`
	"Any X is happy if
	X has any Y and
	Y is a dog."

`likes(X,Y) :-`
`  friends(X,Z),`
`  likes(Z,Y).`
	Any X likes any Y if
	X is a friend of any Z and
	Z likes Y."

Prolog programs are composed of facts and rules. Facts are stored with rules. A collection of facts and rules is commonly referred to as a database. There is no need in Prolog programming to differentiate between programs and data.

Queries. Queries provide a means for retrieving information from a logic program. A query has the special form

$$? - G_1, \ldots, G_n.$$

A query initiates the execution of a program and is typically typed at the terminal with the interpreter running and the desired program loaded. Prolog solves a query by forming a set of variable bindings that makes each of the goals in the query consistent with the facts and rules in the program. This process can be illustrated with the following example:

```
fact 1.   likes(joe,apples).
fact 2.   friend(bonny,joe).

rule 1.   likes(X,Y) :-
            friend(X,Z),
            likes(Z,Y).
```

The interpreter now knows two facts that state that joe likes apples and bonny is a friend of joe. The single rule expresses a more complex relationship. Any X likes any Y if X is a friend of Z and Z likes Y. The database, once loaded, can be interrogated with queries. Consider the following query

```
?- likes(joe,X).
```

The Prolog interpreter will start at the top of the database and search for a fact or rule whose head matches our query. The interpreter knows it is looking for facts or rules named `likes` with two arguments, the first of which must be "joe" or able to be instantiated to

"joe". The second variable X has not yet been instantiated. The first fact encountered that meets all the requirements is fact 1 if X is instantiated to "apples". The interpreter (assuming Prolog-1 is being used) will respond to our query with

```
X = apples
more (y/n)? y
no
```

The interpreter can not find any more facts or rules that it can match with our query.

As a more complex example, consider the compound query

```
?-likes(joe,X),likes(bonny,X).
```

This query asks the interpreter to find some instantiation for the variable X that both "joe" and "bonny" like. The following sequence of events will occur:

1. The first goal of the query succeeds because of fact number 1 with X instantiated to "apples".

2. "bonny" can not be matched to any likes fact so the second goal can only succeed through rule 1. The rule is invoked with X instantiated to "bonny" and the variable Y instantiated to "apples".

3. The first goal of rule 1 succeeds with X instantiated to "bonny" and Z instantiated to "joe" via fact 2.

4. The second goal of rule 1 then succeeds with Z instantiated to "joe" and Y to "apples" using fact 1 again.

5. Since both goals of the query goal succeeded, the query itself succeeds with X instantiated to "apples".

The interpreter's response to our query would then be

```
X = apples
more (y/n)? y
no
```

2.2.2 Programming in Prolog. There are at least three styles of programming languages: procedural, functional, and relational [20].

1. Procedural - In a procedural language, the steps that will eventually produce the desired result are explicitly specified. The programmer must specify, step by step, how something is to be done. The execution of a procedural program is understood by tracing through each step of the process.
2. Functional - In a functional language, methods for calculating values are defined. Operationally, expressions are evaluated until the final value is determined.
3. Relational - A relational language, also known as a logic programming language, specifies relations among values. A logic program can be viewed as a collection of relations with each clause specifying some condition under which the relation holds. Logic programming is declarative: the programmer specifies "what" is to be done, leaving the "how" largely up to the computer.

Prolog programming uses relations within a logical structure to represent knowledge. Deductions, based on the represented knowledge, are used to form logical consequences. Prolog programming allows knowledge to be viewed both procedurally and declaratively. The procedural aspect of knowledge representation is found in the order of execution of the program statements. The declarative aspect is embodied in the logic statements that describe the problem domain and how to solve problems in that domain. The logic programmer is less concerned with how the machine will solve a particular problem and more concerned with the accuracy of the defined relationships of the problem and how they interact and hold under various circumstances. Ideally, the Prolog interpreter takes care of the "how" aspect of problem solving, freeing the programmer to focus on the relations characterizing a problem [45:49].

2.2.2.1 Prolog Syntax. Prolog uses only one data-type, the `term` [7]. Any term has one of three forms:

1. a `constant`,
2. a `variable`, or
3. a `structure`.

A `constant` is either an atom or a number. An atom can be represented either by a sequence of alphanumeric characters beginning with a lower-case letter or by a sequence of characters enclosed in single quotes. A `variable` in Prolog stands for some definite but unidentified object. A variable is expressed by a sequence of alphanumeric characters or underscores, beginning with either an upper-case letter or an underscore. Variables in

Table 2.3. Prolog terms

Constants	Variables
a	A
ab	Ab
a_b	A_B
'AB'	_AB5
1	_1

Prolog do not designate storage-locations in memory as do variables used in conventional languages. Instead, variables are used for pattern-matching within a term. Table 2.3 gives some examples of simple terms.

A compound term is composed of a functor and a sequence of one or more arguments and is commonly referred to as a structure. A structure has the form

$$\underbrace{foo}_{functor}\ \overbrace{(t_1,\ldots,t_n)}^{arguments}$$

where foo, the functor (or predicate), is an atom and $t_1,\ldots,t_n$, the arguments, are terms. The number of arguments is referred to as the arity of the term. The Prolog interpreter distinguishes among the different forms of structures by the name of the functor and its arity. A constant is considered to have an arity of zero. Table 2.4 gives some examples of structures.

Table 2.4. Examples of Prolog structures

Prolog Structure	Functor	Arity
student(name(first(tom))).	student	1
likes(joe,bonny)	likes	2
family(dad(joe),mom(bonny),children(jill,joshua))	family	3

Lists. An important data structure used in logic programming is the list. A list is a sequence of any number of terms such as 6, bob, (fred(man),sally(woman)), and four. Written in Prolog, this list would appear as

[6,bob,[fred(man),sally(woman)],four].

A list is expressed with the following format :

$$[\overbrace{1}^{head},\underbrace{2,3}_{tail}]$$

where the head of a list is the first term (or element) in the list. The head of the list in the above example is the element "1". The tail of a list is itself a list that consists of everything in the original list except the head. The tail of the above example is the list [2,3]. Square brackets "[ ]" are used to denote an empty list. Throughout this thesis a list will be typically represented as [X|Y] where X denotes the head and Y denotes the tail. The list [1,2,3], written in this form would be [1,|[2,3]]. Sometimes it is convenient to denote more than one element at the front of a list. To show this, the representation is [X,Y,Z|Y] where X, Y, and Z are the first three elements of the list and Y is the tail. The list [1,2,3], written in this form would be [1,2,3|[]].

2.2.2.2 Prolog's Search Strategy. A useful concept in the field of artificial intelligence is the search-tree [45:50]. A search-tree is a mapping onto a tree-structure of a search-process. It is instructive to study the search-tree produced by the Prolog interpreter in the execution of a problem. Consider the following program of simple facts.

```
a(1).
a(2).
a(3).
b(0).
b(2).
b(3).
```

and the following query input at the screen prompt:

```
?- a(X),b(X).
```

The query literally translates to "find some instantiation for the variable X such that a(X) and b(X) are true". This problem has two solutions, X = 2 and X = 3. A graphical representation of the search-tree for this problem is found in Figure 2.1. Prolog's built-in search mechanism is depth-first, meaning that it will start at the leftmost branch of the search tree, following it to the bottom before trying a new branch. This is because the interpreter scans the database from top to bottom in an attempt to satisfy the query. In the example, the interpreter is initially interested in finding a fact with the functor "a", as a(X) is the first goal of the query. The first "a" fact encountered in the search is a(1):

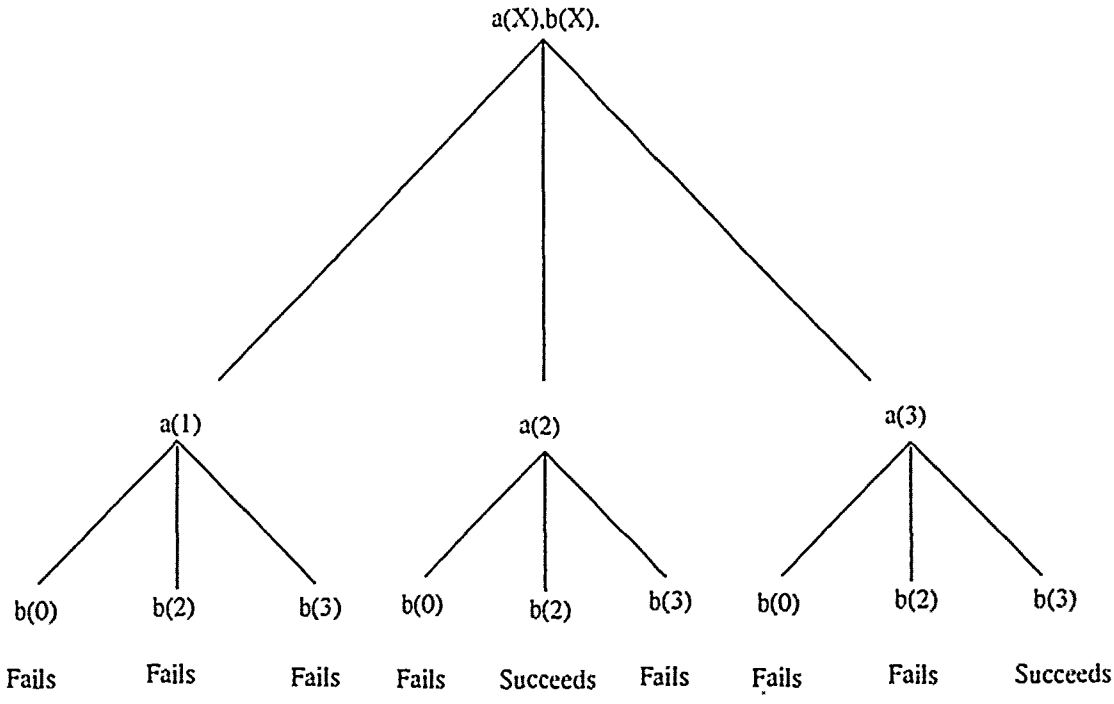


Figure 2.1. A search-tree.

therefore X is instantiated to the value of 1 and the second goal of the query becomes b(1). The interpreter unsuccessfully looks at b(0), b(2), and b(3) before discarding a(1) as a possibility. The interpreter now drops down to the next "a" fact which is a(2). The interpreter picks up a(2) as a possibility for satisfying the query with X instantiated to "2". It then looks at b(0) and finding no match proceeds to b(2) where a successful match is found. At this point the match is reported to the screen and the user asked if more solutions are desired. Typing "no" at the screen prompt will terminate the search. Typing "yes" will cause the search to continue from the point where the interpreter left off. Since a(2) can not be matched with b(3), a(2) is discarded and a(3) is picked up. The interpreter tries to match a(3) with b(0) and b(2) before finding a match with b(3). This second match is reported to the screen as X = 3 and at this point, the database has been searched exhaustively and no more matches exist. The search-tree of Figure 2.1 has been completely traversed from top to bottom, left to right.

2.2.2.3 Control of Program Execution. A conventional algorithm written in either a procedural or functional language is thought to be described as:

program = description of (logic + control)

with the logic component describing the domain-specific part of the algorithm and the control part involving the solution strategy [36:429]. Algorithms written in a procedural or functional language must specify both the logic and the control components: they are intermixed inseparably in the code [33:99]. Algorithms written in a relational language need only specify the logic component of the problem unless, for reasons discussed later, it is advantageous for the programmer to modify the control component.

Prolog only requires the user to provide facts and rules describing relations that are pertinent to the problem domain. Control, often viewed as the "how" of the problem, is ideally left up to the machine. In logic programming, control and logic are separate components of a program and may be specified separately. An algorithm written in a logic programming language can be described as [36]:

$$\text{algorithm} = \text{logic} + \text{control}$$

where the logic and the control components can be disjoint. The order of appearance of the clauses in a logic program should have no logical (declarative) significance because each clause states some property of the predicate independently. This holds true for Prolog as long as control of program execution is left entirely up to the interpreter.

Unmodified Control. As an example of a problem that can be solved by allowing the Prolog interpreter to have full control, consider the task of determining whether an element is a member of a list. Any element X is a member of a list if

```
either (1): X is the head of the list
    or (2): X is a member of the tail of the list.
```

These relations are described in Prolog syntax as follows:

```
rule 1.    member(X,[X|Rest]).

rule 2.    member(X,[Y|Rest]) :-
               member(X,Rest).
```

The name of this procedure is member. The first Horn-clause states that the element X is a member of any list that has X as its head and Rest as its tail. If the element X is the element we are searching for and it occurs at the head of the list then membership is determined and true is returned to the invoking call. If the element X is not the head of the list, the first clause fails and the second clause is invoked. The second clause states the second condition for membership: that is, the element X is a member of a list with an

element Y as its head and Rest as its tail if the element X is a member of Rest. The logic of determining membership is completely specified while the control aspect is left completely unspecified, leaving the interpreter to solve any queries regarding member as it pleases. Note that rule 1 could be switched with rule 2 and only the efficiency of the execution of the procedure would be affected.

Modified Control. Sometimes it can be advantageous to modify the control component in a logic program. Modification is usually done to improve the efficiency of the algorithm. Efficiency can be enhanced by using a special term in Prolog called the "cut." represented by the symbol "!". A cut allows succeeds as a goal and allows a programmer some control over the search mechanism by limiting the interpreter's access to specified branches of the search tree. The effect of the cut limits the number of solutions to the first one found. once the cut has been encountered. Consider this modification to a previous example.

```
a(1).
a(2) :- !.
a(3).
b(0).
b(1).
b(2).
b(3).
```

If the database is queried with the same query as was used previously (?- a(X),b(X).). the interpreter will proceed down the new search-tree, finding the first match between a(1) and b(1). Proceeding on, the interpreter picks up a(2) and encounters the cut. The interpreter tries to match a(2) with b(0) and b(1) as it did in the previous example. before finding a match with b(2). Here is where the difference between the two examples lies. When the second match is reported to the screen, the interpreter will not continue to search for additional solutions. Clearly the database contains one more match, namely X = 3, but the cut effectively "pruned" any other possible match after a(2) from the search-tree. If the first fact of the program (a(1)) were swapped with the second one (a(2) :- !), the search-tree would be even more severely pruned . The execution of the query would result in only one answer, X = 2.

It is important that the modification of the control strategy should only affect the behavior of the computer and not the meaning of the program [36:429]. For example. a programmer may want to go to a local convenience store. If he were to write a logic

program outlining all the methods of transportation available, along with the constraints relating them, the computer could be used to suggest the best method of travel given a set of existing conditions. One such condition might be that it is raining; in such a case, walking would not be appropriate. The control of the program's execution could be altered to exclude exploring any modes of transportation that would expose the programmer to the elements. By doing so, the execution of the program is made more efficient by immediately ruling out entire classes of inappropriate solutions: no time is wasted exploring unfruitful paths.

Importance of Ordering Clauses and Goals. Limitations imposed by the interpreter require the programmer to give careful consideration to the order in which the clauses of a program and the goals of a body are listed. Improper ordering of clauses can affect the efficiency of the search process and, in some cases, cause the program not to execute properly. Consider the rule

```
ancestor(X,Y) :-
  ancestor(X,Z),
  parent(Z,Y).
```

which can be interpreted literally as "X is an ancestor of Y if X is an ancestor of Z and Z is a parent of Y". The ordering of the two goals of the body makes logical sense but when the Prolog interpreter attempts to invoke the procedure, the first goal recursively calls itself, creating an infinite loop. If the two goals in the body are swapped, the logical meaning of the rule is preserved but the interpreter will attempt to satisfy the parent-goal first and an infinite loop will not result. Often, careless ordering of clauses will not cause a program to execute improperly but will affect the efficiency of execution. Efficiency of execution can be affected by

1. the order of appearance of procedures in a program;

2. the order of appearance of clauses in a procedure; and

3. the order of appearance of goals in a rule body.

2.2.2.4 Recursion. The second clause in the `member` example given earlier warrants further discussion as it exploits the concept of `recursion`. Recursion is one of the most powerful features of Prolog because recursion can be exploited to perform repetitive tasks [6]. In ' member example, each time X is not the element at the head of

the list, rule 1 fails..and the interpreter descends another level of recursion until X occurs at the head of the list or Rest is exhausted. Each time rule 2 is invoked, the tail of our list will be successively divided into an instantiation for Y and Rest until the end of the list is encountered. The end of a list is denoted by an empty list ("[]") and is used to form a boundary condition. Using the empty list to form a boundary condition for the member example can be interpreted literally as telling the Prolog interpreter "if you look through the entire list of candidate elements and cannot find an occurrence of X then stop your search because X is not a member of the list". For this particular example, the boundary condition does not need to be stated explicitly because the interpreter will progressively examine the elements of the list starting from the head and working towards the tail. If the end of the list is encountered ([]), there could not have been an occurrence of X in the list; hence, the search fails and the interpreter stops looking. Figure 2.2 shows the execution of the simple question, "is 3 a member of the list [1,2,3]?", formally stated in Prolog as the query `?-member(3,[1,2,3]).`

Most logic problems require the boundary conditions to be explicitly declared in order to avoid infinite loops. If a non_member procedure were desired, the boundary condition would have to be explicitly defined in order to instruct the interpreter when to stop looking. The boundary condition, written in Prolog, would look like

```
non_member(X,[])
```

and would be literally interpreted as "it is true that X is not a member of an empty list". The remainder of the prolog code for non_member is

```
non_member(X,[Y|L]) :-
  X \== Y,
  non_member(X,L).
```

where \== stands for strict inequality.

2.2.3 Summary. Some of the virtues of Prolog have been briefly explained in this chapter. A general summary of the characteristics that make Prolog useful in digital circuit design are [27:16]:

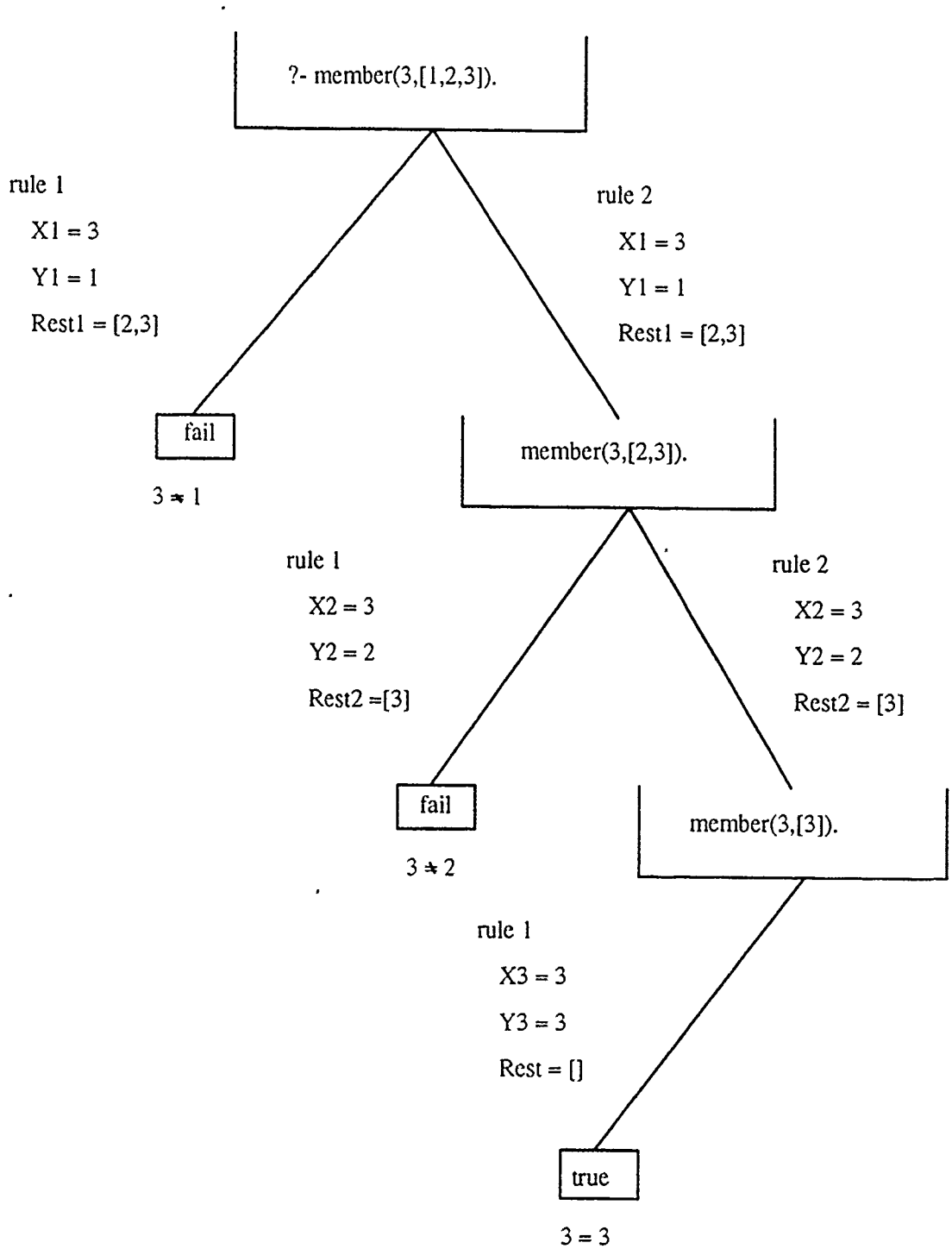


Figure 2.2. Recursion levels for the query `?-member(3,[1,2,3]).`

1. It carries out symbolic inferencing. New facts or rules can be inferred from existing ones.

2. It uses an orderly search process in its execution to find all possible solutions within a problem space.

3. It can be treated as both a declarative and a procedural programming language. The execution of a program may be altered by changing the structure of the logic description of a problem or the sequence of execution or control may be altered.

4. Prolog programs can modify themselves. A program can construct a new fact or rule as it runs and add the fact or rule to itself. Facts or rules can also be retracted from the rule base. This means that Prolog programs can "learn" as they proceed through the execution of a program [19]. An example of this can be found in the the TTL digital circuit simulator presented in Chapter 5. The simulator derives rules concerning the gate-level structure of the circuit under test and uses these new rules to simulate the circuit's operation.

III. Modeling Digital Circuits with Prolog

3.1 Introduction

The versatility of Prolog makes it ideal for modeling logic circuits. Functional and physical characteristics of primitive gates as well as higher-level modules can easily be modeled. Prolog supports the modeling of circuits in a hierarchical fashion. Modeling circuits hierarchically allows lower-level intra-modular connections to be "hidden" from the view of upper-level circuit descriptions. This allows the design engineer to specify the internal connections of a module once as a template and then to reuse the template as needed to design new circuitry.

3.2 Circuit Composition

A circuit is composed of sets of modules and their interconnections. Modules can be defined hierarchically, where modules are defined in terms of other modules. At the bottom of the hierarchy reside the primitive modules whose identity depends on the technology being modeled. Digital designers may consider gates or individual packages to be primitive while a VLSI designer would consider transistors to be primitive [14:61]. Circuits that lack any hierarchical structure are referred to as "flattened" circuits.

3.3 Circuit Representation

There are three basic methods, described in the following paragraphs, used to represent digital circuits [14] in Prolog. The `functional` method uses terms to functionally describe circuits, the `extensional` method uses a database of facts to assert connection relationships, and the `definitional` method uses formulae to model the circuit definitions.

The Functional Method [14:61]. This method can be used to represent circuits in which a single output signal is a function of several input signals [10:391]. Functions can have an arbitrary number of inputs but represent only one output. Inputs are expressed as a module's arguments. Modules, that can be either higher-level or primitive, are named according to the function they perform. For example, the term

$$or(\underbrace{a}_{input}, \underbrace{b}_{input})$$

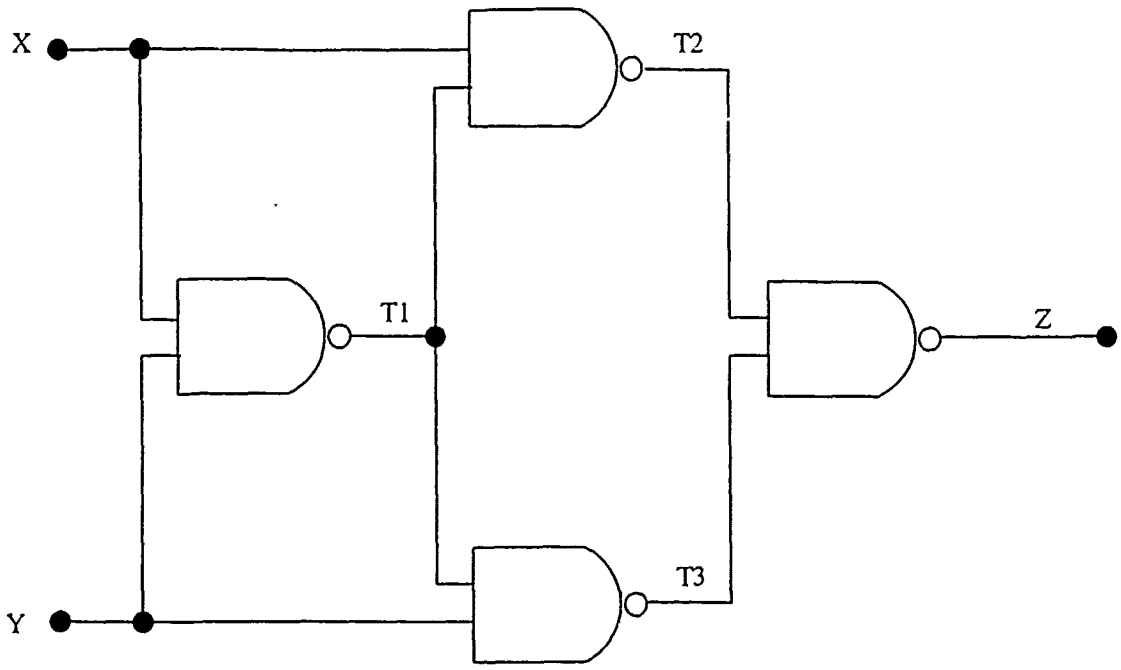


Figure 3.1. XOR-gate function composed of NAND-gates.

would be used to express a two-input OR-gate. Figure 3.1 shows an example of a higher-level circuit constructed of NAND-gates. Using the functional method, the output signal, Z, would be expressed as

```
nand(nand(X,nand(X,Y)),nand(nand(X,Y),Y)).
```

Note that the connectional relationship between modules is purely functional with the syntactic form determining the connections.

This type of representation has two main disadvantages [10:392]. First, only circuits that have no feed-back loops can be described: therefore most sequential circuits can not be accurately modeled with this method. Secondly, each output must be represented with a separate expression. This makes hierarchical representations described with the functional method difficult to understand.

The Extensional Method [14:62] The `extensional` method represents each module and its connections as separate facts. In the example below, the `module` predicate has the following general form:

```
module(Name,List_of_input_ports,List_of_output_ports).
```

The binary predicate `connect` describes the connections between the ports in the following manner:

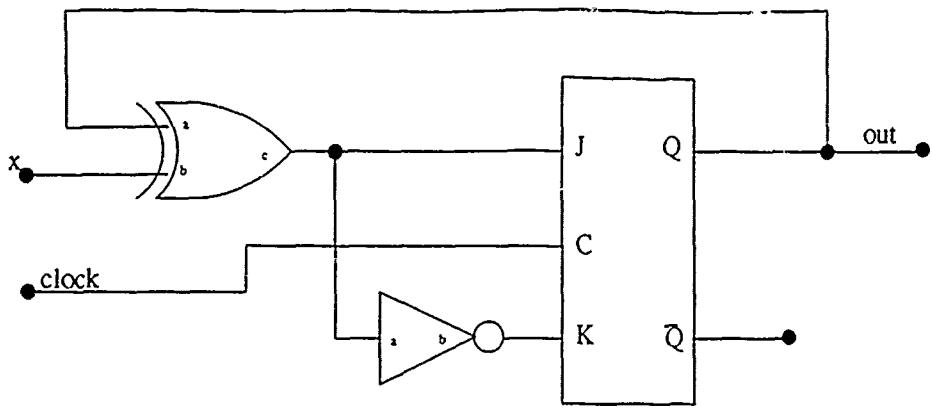


Figure 3.2. Sequential circuit for Extensional example.

```
connect(Specific_port,Connection).
```

Connections between modules are specified with the connect predicate. This predicate has two arguments, the first indicating the module type and the second indicating the connected port [14:62]. Figure 3.2 would be described as

```
module(xor,[a,b],[c]).
module(not,[a],[b]).
module(jkff,[j,c,k],[q,nq]).

connect(xor(a),out).
connect(xor(b),x).
connect(xor(c),cjkff(j)).
connect(xor(c),not(a)).
connect(not(b),cjkff(k)).
connect(clock,cjkff(c)).
connect(jkff(q),out).
```

The module and connect predicates should be considered as generic templates. When the arguments of the predicates are instantiated, the combination of both predicates represents an instance of a circuit component (provided by the module predicate) and a complete description of the component's circuit connections (provided by the connect predicate) [14:63]. Only one module fact is needed for each circuit component to be modeled. However, a large number of connect predicates could be needed to describe the component's circuit connectivity; the exact number required is dependent on the number of other circuit

devices to which the component is connected. Since the simplest component must have at least one input node and one output node, the minimum number of connect facts per component is two. Note that the XOR-gate module described in the code given above needs one module fact and four connect facts to completely describe its relationship in the circuit.

This method can be used to model sequential as well as combinational circuits. However, because the modules are represented with no syntactic relationship between themselves and their connections (one module fact for each module type and two or more connect facts for the connections), some operations such as circuit transformations are hard to implement without resorting to awkward modifications of the database. For example, suppose a logic program were written to identify modules that could be removed from a circuit without affecting the circuit's behavior. Once a redundant module were located, it would need to be identified so the remaining modules in the circuit could be reconnected without the redundant module. If the modules are extensionally defined, there is no way to identify a single module except through the connect predicates. The extra operations necessary to manipulate the structural description in order to compensate for the removed module would be awkward. The simulation of faults would also be awkward if the circuit under test were described extensionally. Often the modeling of a single-gate failure is desired. If there is no syntactic way to describe exactly which individual gate's failure is to be modeled, the simulation can not be carried out. However, despite the inadequacies of the extensional method, it has proven useful in applications related to logic circuit modeling [9, 10, 14].

The expression of modularity is very difficult with either the functional or the extensional methods. In contrast, a method of modeling that employs single terms to describe both a module and its connections can be used to express modularity much more easily. The third method, the definitional method, circumvents the disadvantages of the previous two methods by combining the advantages of both.

The Definitional Method. The definitional method [39:63], used throughout this thesis, represents modules having n ports as n-ary predicates. A Horn-clause is used to describe a module: the head of the clause names the module to be defined and the body names a combination of lower-level and primitive modules. Individual modules within a higher-level module are listed as subgoals of that module; the structure and internal arguments of each subgoal are "hidden" from the higher-level description. This "hiding" of lower-level module descriptions allows the definitional method to be used to describe

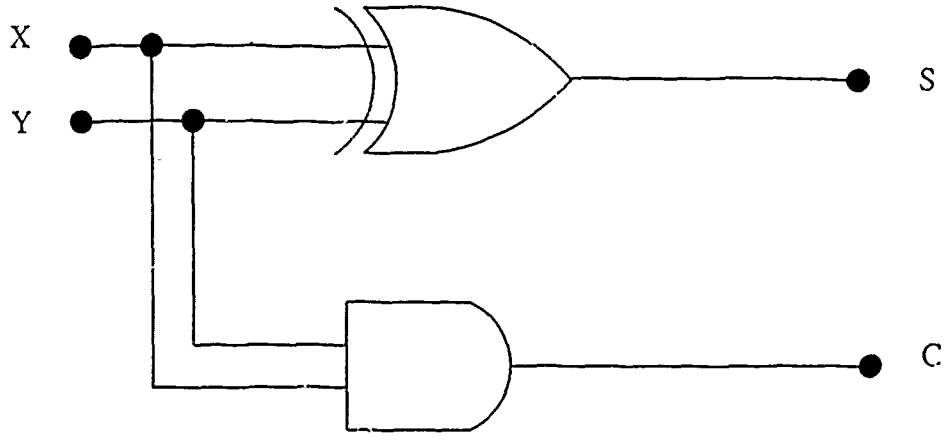


Figure 3.3. Circuit diagram of a half-adder.

circuit organization in a hierarchical fashion without involving extra predicates or arguments. Ports that share a common connection are represented with like-name (coreferred) variables. A half-adder, for example, can be represented hierarchically as shown in Figure 3.3. Note that the XOR function can be represented by the NAND-gate configuration of Figure 3.1.

The definitional method of representing a half-adder easily accommodates the hierarchical nature of the circuit and allows connections to be described in a very understandable manner as illustrated by Figure 3.3 and shown by the following code [13:497]:

```
half_adder(X,Y,S,C) :-
  xor_gate(X,Y,S),
  and_gate(X,Y,C).
```

The predicate xor_gate can be modeled as a higher-level module in terms of primitive NAND-gates (see Figure 3.1) with the following code:

```
xor_gate(X,Y,Z) :-
  nand_gate(X,Y,T1),
  nand_gate(T1,X,T2),
  nand_gate(Y,T1,T3),
  nand_gate(T2,T3,Z).
```

The status of a module (higher-level or primitive) can be stated with a simple primitive predicate and one argument as follows:

```
primitive(nand).
primitive(and).
```

All modules not specified as primitive are assumed to be higher-level.

The definitional method does not explicitly declare the input or output status of a module as did the two methods discussed previously. If it is necessary to know the direction of the ports, another simple predicate can be used. The predicate `direction` can be written explicitly for NAND-modules as

$$direction(\overbrace{nand(A,B,C)}^{name},\underbrace{[A,B]}_{inputs},\overbrace{[C]}^{outputs}).$$

A variation of the definitional method incorporates the direction of the ports directly into the module predicate [31:285]. For example, the predicate `halfadd` can also be written as

```
half_adder(in(X,Y),out(S,C)) :-
  xor_gate(in(X,Y),out(S)),
  and_gate(in(X,Y),out(C)).
```

The definitional method of modeling has several advantages. Most important is the fact that the module name is explicitly part of the specification, permitting easy decomposition. Consider the half-adder of Figure 3.3. Internal connections that are named with variables do not appear in the head of the clause and are effectively hidden as are connections of lower levels in the hierarchy. This allows simple, primitive modules to be easily arranged into more complex circuits without a corresponding increase in the difficulty in comprehending the overall circuit's function. The Prolog code used to represent the higher-level modules is also easy to compose and understand. This method also has several other advantages [14:64]:

1. Prolog can execute circuit simulations directly from module descriptions.
2. An uninstantiated variable is a natural representation of a high-impedance state.
3. Circuits can be manipulated automatically by recursively decomposing their hierarchical structure.

The definitional method has been used with considerable success to model and simulate digital circuits [23, 24, 36]. Complex combinational and sequential circuits have been explored and favorable results reported.

IV. Logic Programming for Circuit-Extraction

4.1 Introduction

A logic circuit designer would normally work with high-level constructs such as registers and buses, seldom descending to the level of gates [7:5]. However, gate-level descriptions do play an important role in design, especially during the layout process. Designing an actual integrated circuit requires a `netlist`. A netlist specifies the type of primitive components (transistors or gates) necessary to realize the desired function, along with their interconnections. A typical netlist is made up of transistors. However, for this discussion, NAND-gates will be considered primitive for simplicity.

While a netlist is suitable for describing circuit structure for the manufacturing process, it is a formless mass of primitive components which can be almost unintelligible to the designer. Circuit-extraction can be used to redefine a circuit's netlist by rearranging a circuit's modular structure into something more easily understood. Extraction removes groups of lower-level components from a netlist and replaces them with a corresponding higher-level component. Extraction can help the designer to understand his own design or reverse-engineer (decipher something already designed) someone else's design by recognizing groupings of lower-level components that constitute a higher-level component.

Prolog is an ideal language to use for the extraction process because extraction is based on the recognition of patterns and Prolog is a pattern-matching language. Only two items are necessary for the interpreter to carry out the extraction process. First, the interpreter needs a netlist of the primitive components that describe the circuit. Second, it needs a description of the pattern or function to be searched for.

In the following example, the interpreter will begin with a netlist composed entirely of NAND-gates. First, any groupings of NAND-gates that perform XOR-gate functions will be found. The appropriate NAND-gates and their connections will be replaced with the higher-level XOR-gate construct with the correct connections. The interpreter will then look for groupings of NAND-gates and XOR-gates that can be replaced with full-adders, again removing the appropriate NAND and XOR-gates with their connections and replacing them with full-adders and the proper connections.

4.2 The Circuit-Extraction Process

Defining XOR-gates in Terms of NAND-gates. Consider the simple netlist (given below) consisting of four two-input NAND-gates described using the definitional

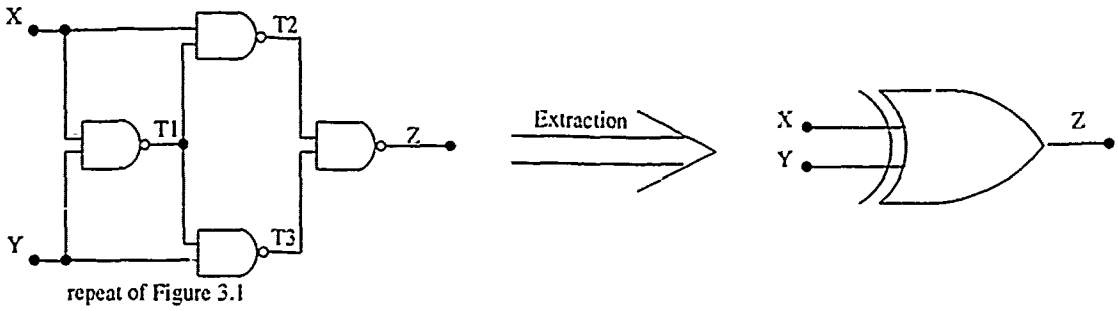


Figure 4.1. Extracting an XOR-gate.

method. Note that the arguments of each fact have been instantiated with a constant representing a nodal position or connection in the circuit.

```
nand_gate(x,y,t1).
nand_gate(x,t1,t2).
nand_gate(y,t1,t3).
nand_gate(t2,t3,z).
```

Once loaded into Prolog's database, the netlist can be scanned and the four NAND-gates recognized as performing the same function logically as a two-input XOR-gate described by

```
xor_gate(x,y,z).
```

Figure 4.1 shows the circuit, before and after extraction.

When the netlist is loaded into Prolog's database, it becomes part of the working program, as data and procedures are not stored separately. Prolog has two built-in procedures that can be used to alter the database by retracting old unwanted facts or rules and asserting new ones.

The built-in predicate `retract(X)` erases a fact or rule from the current database [16:106]. The predicate `retract` takes a single argument `X` that is to match the clause to be retracted. The interpreter proceeds, top to bottom, searching through the database until a match is made. Once the match is made, the matching clause is removed from the database. Only one matching clause is removed each time `retract` is invoked. If it is necessary to remove six clauses (all with the same name), `retract` would have to be invoked six times.

The built in predicate `assert(X)` adds a new clause to the database [16:105]. `assert` comes with options that allow the clause to be put at the beginning of the database

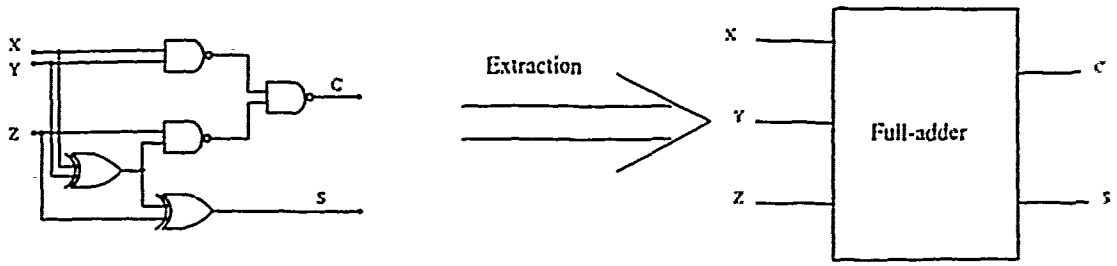


Figure 4.2. NAND and XOR-gate to full-adder conversion.

(asserta(X)) or at the end of the database (assertz(X)). The argument X must be an instantiated clause for assert to succeed.

Defining a Full-Adder in Terms of XOR and NAND-gates. A full-adder is a device that has three input-terminals, each carrying a single binary bit. Functionally, a full-adder counts the number of its inputs that have a value of 1. The sum of the inputs is represented with two output-terminals as a two-digit binary number, the carry-bit C and the sum-bit S. If two of the three inputs have a value of 1 then the output would be the number CS = 10, which is the binary representation of the decimal value 2.

A full-adder can be defined in terms of NAND and XOR-gates (see figure 4.2). Consider the following netlist, in which each argument has again been instantiated to a constant that represents a circuit-node or connection point.

```
xor_gate(x,y,t1).
nand_gate(x,y,t2).
xor_gate(z,t1,s).
nand_gate(z,t1,t3).
nand_gate(t2,t3,c).
```

Once loaded into prolog's database, this netlist can be subjected to the extraction process and the five gates recognized as constituting a full adder described by

```
full_adder(x,y,z,s,c).
```

Figure 4.2 shows the circuit before and after extraction. After the Prolog interpreter recognizes the pattern for a full-adder in the netlist, the five gates are retracted from the netlist and the replacement device asserted to fill the void.

4.3 The Circuit-Extraction Algorithm

The process to extract full-adders can be formally stated as

1. Load the netlist to be scanned into the database.
2. Scan the netlist for all possible XOR-gate instances of NAND-gate configurations. When one is identified, retract the NAND-gate constituents and assert the replacement XOR-gate.
3. Scan the revised netlist for all possible full-adders. Retract the appropriate configurations of XOR and NAND-gates and replace them with full-adders.

NAND-gate configurations that can be replaced by XOR-gates can be found for particular instances with the following code.

```
xor_gate(X,Y,Z) :-
  nand_gate(X,Y,T1),
  nand_gate(X,T1,T2),
  nand_gate(Y,T1,T3),
  nand_gate(T2,T3,Z).
```

When compared against a netlist, Prolog will instantiated the upper-case letters, which represent variables, with the netlist's lower-case letters which represent actual nodal connections. The xor_gate clause above can be reused as often as is necessary to completely scan the netlist and can be considered a generic template that describes the general connectivity of an XOR-gate to the interpreter. If the interpreter can find nodal connections within the circuit definition that fit the pattern of the XOR-gate template, the substitution will take place. If the interpreter matches the variables in three of the four NAND-gate goals but fails to find a match for the fourth, it will "undo" the first three goal's instantiations and proceed with another combination until all possible permutations are tried. This is done automatically because of Prolog's backtracking ability. In an actual extraction program the goals of the xor_gate clause must be modified slightly to account for a transposing of the input values. Each goal of the clause should succeed without regard to the order of the input value arguments. For example, the goal nand_gate(X,Y,T1) should be accompanied by the goal nand_gate(Y,X,T1); thus informing the interpreter that either arrangement of input arguments is acceptable. The remaining three goals of the xor_gate clause require similar companion-goals in order to completely describe all the possible permutations of input arguments that could occur.

In a similar manner, the complete code for defining a full-adder is

```
full_adder(X,Y,Z,S,C) :-
  (nand_gate(X,Y,T2)
  ;
   nand_gate(Y,X,T2)),
  (xor_gate(X,Y,T1)
  ;
   xor_gate(Y,X,T1)),
  (nand_gate(Z,T1,T3)
  ;
   nand_gate(T1,Z,T3)),
  (xor_gate(Z,T1,S)
  ;
   xor_gate(T1,Z,S)),
  (nand_gate(T2,T3,C)
  ;
   nand_gate(T3,T2,C)).
```

where the symbol ; denotes a disjunction.

4.4 An Abbreviated Extraction Session

In addition to the code discussed previously in this chapter, a netlist of components and a procedure to drive the extraction process are needed to successfully start a sample extraction session. The code necessary to drive the process is

```
extract_full_adders :-
   find_xor_gates,
   find_full_adders.
```

The procedures `find_xor_gates` and `find_full_adders` describe the pattern of NAND-gates that constitute an XOR-gate or a full-adder respectively. A complete listing of the code is given in Appendix A.

Any netlist can now be subjected to the extraction process. The following netlist is provided for this example:

```
nand_gate(x,y,t1).
nand_gate(y,t1,t3).
nand_gate(x,t1,t2).
nand_gate(t2,t3,t4).
nand_gate(t4,z,t1).
nand_gate(t4,t1,t2).
nand_gate(z,t1,t3).
nand_gate(t2,t3,s).
nand_gate(x,y,t6).
nand_gate(z,t4,t5).
nand_gate(t6,t5,c).
```

which describes exactly one full-adder.

Running the extraction produced the following short script session. A more in depth session is given in Appendix A. Comments and notes that are not a part of the actual script session are set off with the standard Prolog1 comment delimiter /*.....*/.

```
Script V1.0 session started Mon Sep 02 12:44:22 1991

Microsoft(R) MS-DOS(R) Version 4.01
             (C)Copyright Microsoft Corp 1981-1988

J:\THESIS\CODE>prolog

+-----------------------------------------+
| MS-DOS Prolog-1             Version 2.2 |
| Copyright 1983      Serial number: 0001213 |
| Expert Systems Ltd.                     |
| Oxford U.K.                             |
+-----------------------------------------+
```

/* The following command loads the file extract.pro. The file contains all the Prolog code necessary for this session. */

```
?- [extract].
extract consulted.
```

/* invoking the algorithm. */

```
?- extract_full_adders.
yes
```

/* The built-in predicate "listing(X)" will find all expressions for X and print them on the screen. */

```
?- listing(nand_gate).
yes

/* All NAND-gates are gone. */

?- listing(xor_gate).
yes

/* All XOR-gates are gone. */

?- listing(full_adder).
full_adder(x,y,z,s,c) .
yes

/* One full-adder, with the proper nodal connections, was found. */

?- halt.

/* Finished . */

J:\THESIS\CODE>exit
Script completed Mon Sep 02 12:46:22 1991
```

There are several ways that the procedure could be made more efficient, but this would adversely affect the readability of the code. Efficiency has been sacrificed for ease of understanding. For example, one helper procedure could be called to find the XOR-gate configurations and another helper procedure called to retract the NAND-gates and assert the XOR-gate. However, since variables are only known within the scope of the applicable procedure, additional parameters would have to be set up and passed.

V. *The Simulation of Digital Circuits with Logic Programming*

5.1 Introduction

A digital logic circuit can be viewed as a network of modules, both higher-level and primitive, whose interconnections impose constraints on the circuit. Satisfying those constraints with some connections bound to constant values can serve to simulate the operation of the circuit [31:283]. Conventional modeling programs like `Spice` are restricted to quantitative modeling in the "forward" direction only [31:284]. Circuit modeling with Prolog does not suffer from the same limitations, because of the bidirectional nature of logic programming [23]. By specifying a subset of the inputs to a circuit under test, Prolog can provide all possible corresponding output values. If a subset of the output values is specified, Prolog can provide all input values that are possible under those circumstances. If a combination of input and output values is specified, Prolog will find all possible combinations of unspecified input and/or output values.

Unlike the extraction process of the previous chapter, the simulation of digital circuitry does not involve the manipulation of nodal connections. Instead, the variables in the circuit definitions are instantiated with Boolean values, either 0 or 1. The convention for naming functors to reflect this difference is discussed in Chapter 1.

5.2 The Simulation of Combinational Circuits

5.2.1 Primitive Modules. The simulation of the operation of combinational circuits composed of primitive modules is easily accomplished with Prolog. In a combinational circuit, the value of the output depends only on the present value of the input. As an example, consider a circuit that contains one two-input NAND-gate. At any time, each of the input signals may be at one of two voltage levels; these are represented by the Boolean values 0 and 1. The output of the NAND-gate is at level 0 if and only if all its inputs are at level 1; otherwise the output is at level 1. This behavior is defined by the truth-table shown in Table 5.1 where `X` and `Y` represent input signals and `Z` represents the output signal. The operation of a two-input NAND-gate can be described in Prolog with four facts, one for each row of the Truth-table (Table 5.1).

```
fact 1  nand(0,0,1).
fact 2  nand(0,1,1).
fact 3  nand(1,0,1).
fact 4  nand(1,1,0).
```

Once the four facts describing the operation of a NAND-gate are supplied to the Prolog interpreter, the gate's operation can be simulated forward with the following query:

```
?- nand(0,0,Out).
   Out = 1
   more (y/n)? y
   no
```

The query, directly above, asks "if the two input signals of a two-input NAND-gate are set to the Boolean value 0, what possible values can the output variable Out hold ?". The NAND-gate truth-table shows that only one value exists for the given combination of inputs, Out = 1. Simulation in the backward direction is also possible. The query

```
?- nand(X,1,Out).
```

will yeild the following response from the interpreter.

```
X = 0
Out = 1
more (y/n)? y


X = 1
Out = 0
more (y/n)? y
no
```

Table 5.1. NAND-gate truth-table.

X	Y	Out
0	0	1
0	1	1
1	0	1
1	1	0

Table 5.2. XOR-gate truth-table.

X	Y	Out
0	0	0
0	1	1
1	0	1
1	1	0

5.2.2 Higher-Level Modules. In order to simulate the operation of a higher-level module, it must be defined in terms of lower-level modules. At the bottom of the module hierarchy are the primitive submodules. Consider an XOR-gate. The output of an XOR-gate produces the modulo-2 sum of all its inputs [7:2]. Since the range of the input values is limited to either the Boolean value 0 or 1, the operation of an XOR-gate may be thought of as counting the number, N, of the XOR-gate's 1-inputs. If N is odd, the output signal level is 1; if N is even, the output is 0. Table 5.2 defines this behavior for a two-input XOR-gate.

In order to simulate the operation of an XOR-gate, two approaches are possible. First, the gate could be declared as a primitive gate and the values in Table 5.2 listed as facts in the database, similar to the procedure described for the NAND-gate or second, the XOR-gate function could be modeled hierarchically in terms of other modules. Taking the latter approach, one of many possible ways to construct a digital circuit that behaves as an XOR-gate is shown in Figure 3.1. This hierarchical representation can be simulated if the constituent NAND-gates are declared primitive and the NAND-gate truth-table (see Table 5.1) entered into the database. All that would be needed to initiate a query would be the rule that defines the XOR-gate hierarchically in terms of NAND-gates. The rule, coded in Prolog, would be

```
xor(X,Y,Z) :-
  nand(X,Y,T1),
  nand(X,T1,T2),
  nand(Y,T1,T3),
  nand(T2,T3,Z).
```

When the simulator is presented with the query

```
?- xor(1,1,Z).
```

the following actions take place:

1. The first goal of the XOR rule, nand(X,Y,T1) has both X and Y instantiated to the value of "1" because of the format of the query. Nand-fact 4 (see page 5-2) can satisfy the first goal if T1 is instantiated to the value of "0". With this done, the second goal now reads nand(1,0,T2).
2. Nand-fact 3 can satisfy the second goal if T2 is instantiated to the value of "1". With this done, the second goal is now satisfied and the interpreter attempts to satisfy the third goal, nand(1,0,T3).
3. The third goal can be satisfied through the application of nand-fact 3 also. This instantiated T3 to the value of "1". The interpreter now moves on to the fourth goal, nand(1,1,Z).
4. Nand-fact 4 can be used to satisfy the fourth goal of the XOR rule resulting in Z being instantiated to 0. The interpreter has now found one possible solution and correctly reports Z = "0" as the result. In this example, no more solutions are possible as no more nand-facts can be used to satisfy the XOR rule's goals.

Applying this approach, Prolog allows the operation of very complex hierarchically-designed circuits to be simulated with ease. For circuits having large numbers of input lines, the simulation of all modes of operation is not practical; if done with discretion, however, simulation does provide the designer with a reasonable degree of confidence in his design [7:4]. The following bidirectional simulation of a four-bit adder serves to illustrate the usefulness of the simulation of complex digital circuitry with Prolog.

5.2.2.1 Defining a Full-Adder in Terms of XOR-Gates and NAND-Gates. A full-adder is a device that has three input-terminals and two output-terminals. The range of the input and output values is limited to the Boolean values of 0 or 1 just as was done in the previous examples. The outputs are labeled (by convention) C (carry) and S (sum). The operation of a full-adder is similar to that of an XOR-gate in that it counts the number, N, of its inputs that have the value 1. The value of N is displayed at the output of the full-adder as a two-bit binary number. For example, if all three inputs have the value 1, then the output would be CS = 11, or binary 3. A full-adder was defined in terms of XOR- and NAND-gates in the previous chapter. The schematic representation of a full-adder is shown in Figure 4.2. Each XOR-gate in the full-adder can be modeled by four NAND-gates; hence a flattened representation of a full-adder could contain as many as 11 two-input NAND-gates. The conventional symbol for a full-adder is shown in Figure 5.1.

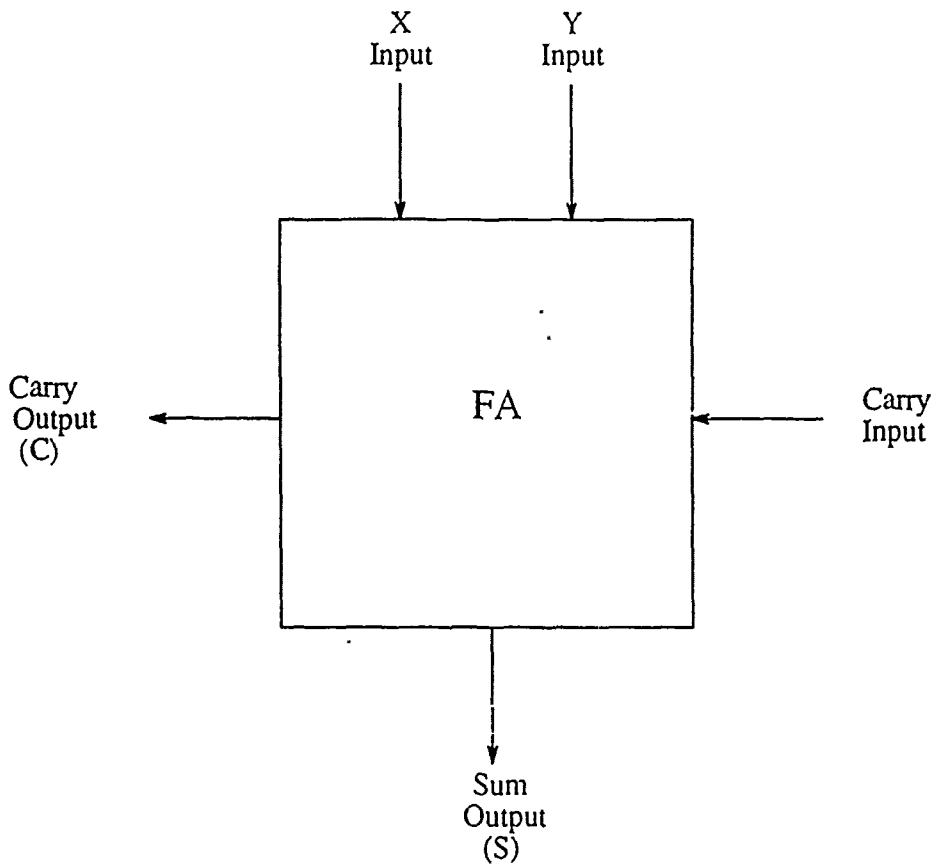


Figure 5.1. Conventional representation of a full-adder.

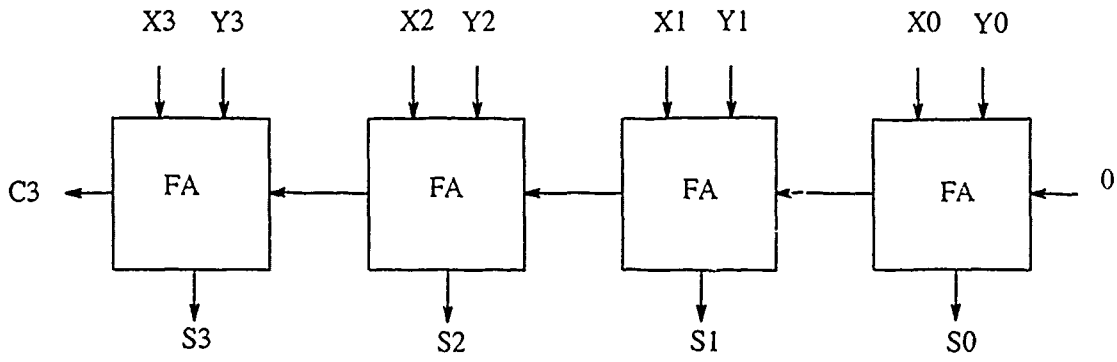


Figure 5.2. Four-bit binary adder constructed from full-adders.

5.2.2.2 Defining an n-Bit Binary-Adder in Terms of Full-Adders. An n-bit binary adder works in a fashion similar to that of the full-adder. The n-bit binary adder produces a binary sum represented by n+1 output bits given two n-bit binary numbers as inputs. Figure 5.2 shows a four-bit binary-adder.

Consider the following netlist

```
full_adder(X0,Y0,0,C0,S0).
full_adder(X1,Y1,C0,C1,S1).
full_adder(X2,Y2,C1,C2,S2).
full_adder(X3,Y3,C2,C3,S3).
```

that models the circuit shown in Figure 5.2. Note that the value of the signal on the input carry line has been set to 0 for convenience. The netlist's operation can be simulated by defining its head functor and arguments as

```
four_bit_adder(bin(X3,X2,X1,X0),bin(Y3,Y2,Y1,Y0),bin(C3,S3,S2,S1,S0)).
```

Note that there is no actual procedure named bin. The functor bin is used for pattern-matching and allows the arguments of the adder to be grouped into a more recognizable form. Also note that the functor, bin, does not have the same number of arguments in all cases. Prolog will recognize this and treat the functors with different numbers of arguments as different functors. The complete Prolog code that describes the operation of a four-bit adder is

```
four_bit_adder(bin(X3,X2,X1,X0),bin(Y3,Y2,Y1,Y0),bin(C3,S3,S2,S1,S0)) :-
  full_adder(X0,Y0,0,C0,S0),
  full_adder(X1,Y1,C0,C1,S1),
  full_adder(X2,Y2,C1,C2,S2),
  full_adder(X3,Y3,C2,C3,S3).
```

It is interesting to note that a flattened four-bit binary adder would require 44 NAND-gates. Construction and simulation of a 44 gate circuit is considerably more difficult if advantage of the hierarchical structure is not taken.

5.2.3 An Example of a Four-Bit Binary Adder Simulation. The simulation of the operation of a four-bit binary adder, modeled hierarchically with NAND-gates, can be accomplished with the code discussed above. The required code is repeated in a consolidated form in file addrcode.pro located in Appendix B. Comments, that are added to the simulation session for clarification, are set of with the standard Prolog-1 comment syntax ./*.....*/.

```
Script V1.0 session started Mon Sep 09 09:24:35 1991

Microsoft(R) MS-DOS(R) Version 4.01
             (C)Copyright Microsoft Corp 1981-1988

J:\>prolog
+-------------------------------------------+
| MS-DOS Prolog-1              Version 2.2  |
| Copyright 1983       Serial number: 0001213  |
| Expert Systems Ltd.                       |
| Oxford U.K.                               |
+-------------------------------------------+
```

/* This command loads the file which contains the code necessary for the simulation to take place. */

```
?- [addrcode].
addrcode consulted.
```

/*Test with all inputs set equal to zero. */

```
?- bin_adder(bin(0,0,0,0),bin(0,0,0,0),bin(C3,S3,S2,S1,S0)).

C3 = 0
S3 = 0
S2 = 0
S1 = 0
S0 = 0
More (y/n)? y

no
```

/*Reverse simulation test. Given that all X inputs are set to zero and the output is given as binary 1, what must value at Y have been ? */

```
?- bin_adder(bin(0,0,0,0),Y,bin(0,0,0,0,1)).

Y = bin(0,0,0,1) /*correct value*/
More (y/n)? y

no
```

/*With the output set at 1, find all possible input combinations. */

```
?- bin_adder(X,Y,bin(0,0,0,0,1)).
X = bin(0,0,0,0) /*one of two*/
Y = bin(0,0,0,1)
More (y/n)? y
X = bin(0,0,0,1) /*two of two*/
Y = bin(0,0,0,0)
More (y/n)? y

no
?- halt.

J:\>exit
Script completed Mon Sep 09 09:31:51 1991
```

More detailed simulation results for a four-bit binary adder along with a complete listing of the associated Prolog code are included in Appendix B.

5.3 The Simulation of Sequential Circuits

Sequential circuits are more difficult to simulate than combinational circuits because any output depends on both the present and past input values [41:949]. A sequential network effectively has a "memory" because it must remember something about the past

sequence of inputs. A combinational circuit has no such memory. A sequential network is composed of a combinational network and some added memory elements. The memory function in sequential networks is performed by memory devices, some of which are flip-flops, counters, and shift-registers. Since flip-flops are the most basic memory device, this discussion will focus on simulating sequential circuits, all of whose memory elements are flip-flops.

One of the most widely used members of the flip-flop family is the JK flip-flop. A JK flip-flop can be represented with the with the relation

```
jkff(J,K,Ps,Ns).
```

where J and K are input terminals, Ps is the present state and Ns the next state. To create a program that simulates the operation of a JK flip-flop, the complete operation of the flip-flop must be specified and the appropriate code added to the database. The following code, derived from Table 5.3, specifies the operation of the JK flip-flop. Since all JK flip-flops are clocked on either the leading or trailing edge of the clocking pulse, actual JK flip-flops must be supplied with an adequate clocking pulse in order to operate properly. However, the simulation of flip-flop operation is possible without specifying any input clock pulse because each clock cycle is represented as one recursive call to the simulation procedure. During any one cycle, all next-state values Ns are found from the combination of present-state values, J, K, and Ps. Realizing that a clock pulse is superfluous allows the JK flip-flop to be modeled without specifying a clocking sequence. The code that describes the complete operation of a JK flip-flop is

```
jkff(0,0,0,0).
jkff(0,0,1,1).
jkff(0,1,0,0).
jkff(0,1,1,0).
jkff(1,0,0,1).
jkff(1,0,1,1).
jkff(1,1,0,1).
jkff(1,1,1,0).
```

Since the operation of a flip-flop requires the specification of an input sequence of J, K, and Ns values, recursion is used to simulate sequential operation. Simple recursive routines are necessary in order to "feed" the input values to the device in a serial fashion. A verbal description of the algorithm necessary to simulate JK flip-flop operation would

Table 5.3. State transition table for a JK flip-flop.

J	K	Ps	Ns
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

be:

1. Using the Heads of the J and K input lists, and the present state Ps, find the value of the next state Ns.
2. Continue the simulation with the tails of the J and K lists. Use the value of the next state Ns found in the previous step as the value of the present state Ps for each iteration.
3. When the lists of inputs are exhausted, return a list of all the next state, Ns, values.

The Prolog code that accomplishes this begins with the boundary condition specified as

```
jksim([],[],_,[]).
```

foliowed by the recursive procedure

```
jksim([J|Jr],[K|Kr],Ps,[Ns|Nr]) :-
  jkff(J,K,Ps,Ns),
  jksim(Jr,Kr,Ns,Nr).
```

5.3.1 A Flip-Flop Simulation Session. The Prolog code listed immediately above and the code that describes the JK flip-flop's operational characteristics can be combined to simulate the operation of a single device. A fully commented listing of the code required to perform this simulation is given in Appendix B under the file name jkcode.pro. Appendix B also contains a detailed simulation session in verbose form. A smaller and less verbose extract of the simulation is presented here.

```
Script V1.0 session started Tue Sep 10 11:40:29 1991

Microsoft(R) MS-DOS(R) Version 4.01
            (C)Copyright Microsoft Corp 1981-1988

A:\THESIS\CODE>prolog
+-----------------------------------------+
| MS-DOS Prolog-1             Version 2.2 |
| Copyright 1983      Serial number: 0001213 |
| Expert Systems Ltd.                     |
| Oxford U.K.                             |
+-----------------------------------------+
```

/* This command loads the file which contains the code necessary for this simulation. */

```
?- [jkcode].
jkcode consulted.
```

/* Starting with a simple test case. */

```
?- jksim([0],[0],0,Q).
Q = [0]
More (y/n)? y
no
```

/* Testing a median case. */

```
?- jksim([1],[0],0,Q).
Q = [1]
More (y/n)? y

no
```

/* Testing an end condition. */

```
?- jksim([1],[1],1,Q).
Q = [0]
More (y/n)? y

no
```

/* Testing an input sequence. */

```
?- jksim([1,0,1,1,1],[0,1,1,1,0],0,Q).
Q = [1,0,1,0,1]
more (y/n)? y
```

```
no
?- halt.

A:\THESIS\CODE>exit
Script completed Tue Sep 10 11:45:50 1991
```

Circuits that contain any number of combinational devices and more than one flip-flop can be very difficult to accurately simulate. The difficulty lies in the additional constraints placed on the timing aspect of the circuit If the proper signal is not at the input terminals of a flip-flop at the correct time, the device could enter an incorrect state, resulting in an erroneous output. These timing conflicts are commonly referred to as race conditions. Race conditions are avoided by using edge triggering devices or master-slave configurations in actual circuit implementations. In an actual circuit, all devices are triggered concurrently. The simulation of this concurrent triggering action cannot be accomplished with Prolog because of the sequential nature in which the interpreter executes the goals of a procedure.

Simulation-Races. Precautions must be taken to avoid `simulation-races` when simulating the operation of complex sequential circuits with Prolog. The circuit shown in Figure 5.3 might be improperly modeled with the following code:

```
jkcrt([],[]_,_,[]).

jkcrt([X|Xr],[Y|Yr],Ps_1,Ps_2,[Ns_2|Rest]) :-
  jkff(X,Y,Ps_1,Ns_1),
  jkff(Ns_1,Y,Ps_2,Ns_2).
```

with the operation of the JK flip-flop specified by the same set of facts used in previous examples. Ps_1 and Ps_2 specify the present state of the respective flip-flop. If the database is queried with

```
?- jkcrt([1],[0],0,0,Q).,
```

the interpreter will respond with

```
Q = [1]
more (Y/N) y
no
```

Inspection of Figure 5.3 shows that the correct output for the given query is `Q = [0]`. The error is due to the simulation-race that exists between the two devices of the circuit. Since

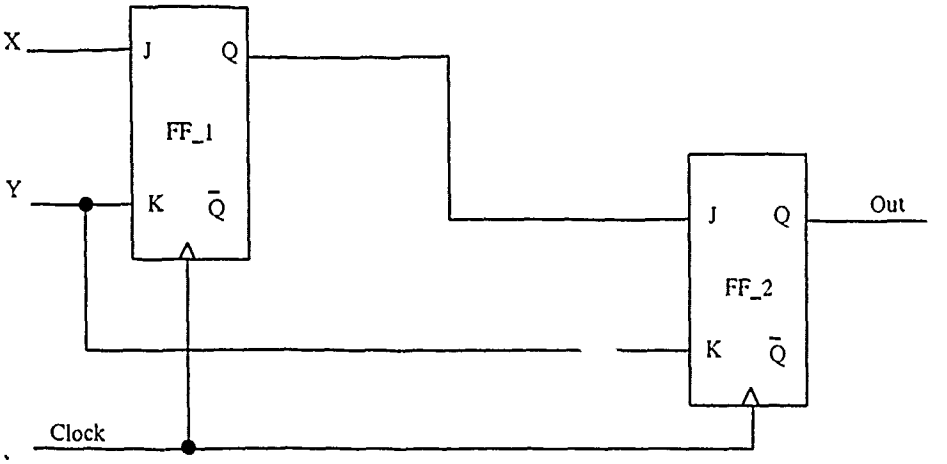


Figure 5.3. Sequential circuit.

the operation of the flip-flops is simulated sequentially, each device is effectively clocked sequentially instead of in parallel as would be done in an actual circuit. The simulation effectively "clocked" the first flip-flop, `FF_1`, and produced the resulting next state `Ns_1`. The new value for `Ns_1` was passed on to the second flip-flop `FF_2`, altering the value on its input J terminal before the simulation had a chance to "clock" the old value on the J terminal through.

Simulation-races can be avoided by "decoupling" the output terminals of the flip-flops from any devices that utilize flip-flop outputs. The simulation is allowed to proceed with present state values and the next state values are passed recursively on to become the next simulation's present state values. Rewriting the code given above for the predicate `jkcrt` to reflect this change gives

```
jkcrt([X|Xr],[Y|Yr],Ps_1,Ps_2,[Ns_2|Rest] :-
  jkff(X,Y,Ps_1,Ns_1),
  jkff(Ps_1,Y,Ps_2,Ns_2).
```

that shows the altered relationship between input and output values necessary to prevent a simulation-race from developing.

5.4 TTLS - A TTL Digital Circuit Simulator

The previously discussed methods of simulating the operation of combinational and sequential digital circuits can now be consolidated to form a general purpose simulator. Since most engineers have some experience with standard transistor-transistor logic (TTL)

design, a simple-to-use TTL circuit simulator can be a very useful design tool. With this in mind, the Transistor-Transistor Digital Logic Circuit Simulator (TTLS) was designed. TTLS is a pin-level logic circuit simulator that can be used to test the performance of digital circuits constructed from a restricted set of 7400-series TTL integrated circuits. The operation of any TTL circuit can be simulated with TTLS as long as the circuit's constituent integrated circuits are listed in the TTLS database file (ttldata.pro).

5.4.1 Key Features of TTLS. TTLS employs several features that separate it from other reported digital circuit simulators [31, 36, 41].

Stack Instantiation. TTLS uses "stack-instantiation" to unify variables. Structures with partially instantiated arguments are arranged on a Prolog goal stack in a manner that invokes a circuit-rule defining the desired circuit operation. When the circuit-rule "fires," all uninstantiated variables on the stack are unified. The instantiated structures are removed from the stack and reported to the user as the simulation results. Stack-instantiation allows circuit simulation to proceed at a very quick pace.

Memory Devices. TTLS can be used to simulate the operation of circuits that use memory devices such as flip-flops. All memory devices are automatically cleared before a simulation sequence is started. Present state and next state values are carried recursively, through the simulation, as arguments. These arguments, as well as other functions necessary to simulate a digital circuit's operation, are managed internally by TTLS and are not visible to the user.

Built-in User Interface. TTLS has a built-in user interface that guides the user through the simulation process. The interface prompts the user for any additional data required for the simulation to take place. After the initial simulation is completed, TTLS prompts the user either to continue the simulation with a new input sequence or to quit.

Built-in Self-Tests. TTLS is written in Prolog-1 and is designed to make use of Prolog's pattern-matching capabilities. Several key points in the execution of the simulation are tied directly to Prolog's ability to detect an error if patterns should match but do not. For example, if the user attempts to run a simulation with a circuit requiring three input values per clock cycle but he/she only specified two values, the Prolog interpreter cannot unify the erroneous input pattern and an error message informs the user of the problem and allows him/her to correct the mistake.

The Prolog interpreter also knows that each integrated circuit output pin being used must be connected to input pins. The TTL data file specifies exactly how many input pins

should correlate to an output pin for a particular package. Although the interpreter never explicitly counts the exact number of input pins defined in the wire-list for each output pin, it can detect an input pin inadvertently left unconnected (floating). If the interpreter does detect a floating input pin, an error message is generated.

The TTLS Process. TTLS derives a gate-level circuit definition from the user provided input data file and asserts the definition into the database as a fully executable Prolog circuit-rule. The circuit-rule is derived in an uninstantiated form and as such is fully reuseable. The predicate head of the circuit-rule is structured in such a way that new simulation sessions are initiated by reconfiguring a copy of the predicate head (called a test-head) for each simulation clock cycle, this avoiding the computational overhead associated with redefining the circuit-rule for each cycle. The structure of the test-head allows related data, such as input and output values, to be manipulated as related objects rather than individual elements. The head is configured with lists as arguments and has the form

```
circuit(ListOfInputValues,ListOfOutputValues,ListOfMemoryDeviceArgs).
```

with the functor "circuit". The first list, `ListOfInputValues`, represents the input values for a particular clock cycle. At the start of a simulation cycle, the old list of input values (if there is one) is removed from the head and the new list of input values is inserted. The list of output values, `ListOfOutputValues`, enters into the simulation as uninstantiated arguments and after exposure to the goal stack, becomes fully instantiated. When fully instantiated, `ListOfOutputValues` contains one clock cycle's simulation results. The list of memory device arguments, `ListOfMemoryDeviceArgs`, is invisible to the user. TTLS automatically sets this list up and manages it through the simulation process in order to propagate memory device state values from one cycle to another. `ListOfMemoryDeviceArgs` is empty if there are not any memory devices in the circuit. Manipulating lists instead of individual elements contributes to TTLS's low execution time.

5.4.2 The Circuit-File. In order to initiate a simulation session, the user must provide an input data file, referred to as the circuit-file, containing the following:

1. a wire-list,
2. a list of integrated circuits,
3. a sequence of input values.

The circuit-file can be named as the user pleases but must end with a ".pro" extension. Any editor capable of creating a file in ascii format can be used to create the circuit-file. Each of the three constituent parts of a circuit-file is further described below.

The Wire-List. The wire-list describes all of the test circuit's inter- and intra-connections, including input and output connections. The wire-list must be specified using the following format:

```
wire_list :-
  [InputA,dic_num,dic_pin_num],
                .
                .
                .
  [WireNum,sic_num,sic_pin_num,dic_num,dic_pin_num],
                .
                .
                .
  [OutputNum,sic_num,sic_pin_num].
```

where `WireNum` is the wire name expressed as a Prolog variable. This may be any name the user desires and may be connected to as many destination integrated circuits as desired. As a note of caution, TTLS will allow any integrated circuit's output pin to be connected to an unlimited number of input pins; therefore, it is incumbent on the circuit designer to avoid fan-out problems when actually implementing a circuit simulated with TTLS.

The source integrated circuit number, `sic_num`, is a user-defined number for the particular integrated circuit (ic1,ic2,...icN). The number `sic_pin_num` is the source integrated circuit pin number and is the actual pin number of the source integrated circuit (sic_num) to which the wire, WireNum, is connected. The other end of WireNum is connected to the destination integrated circuit, `dic_num`, at the destination integrated circuit pin number `dic_pin_num` (see Figure 5.4).

Input and output connections are specified in the format of the first and last `wire_list` entry in the example above. Input and output connections need only a wire-name, expressed as a Prolog variable, and a destination integrated circuit and pin number. There is no limit to the number of input or output signals that can be specified. However, input and output signal-wires must be declared in a fashion that directly correlates the specified input or output value sequence to the order of their listing in the wire-list. For example,

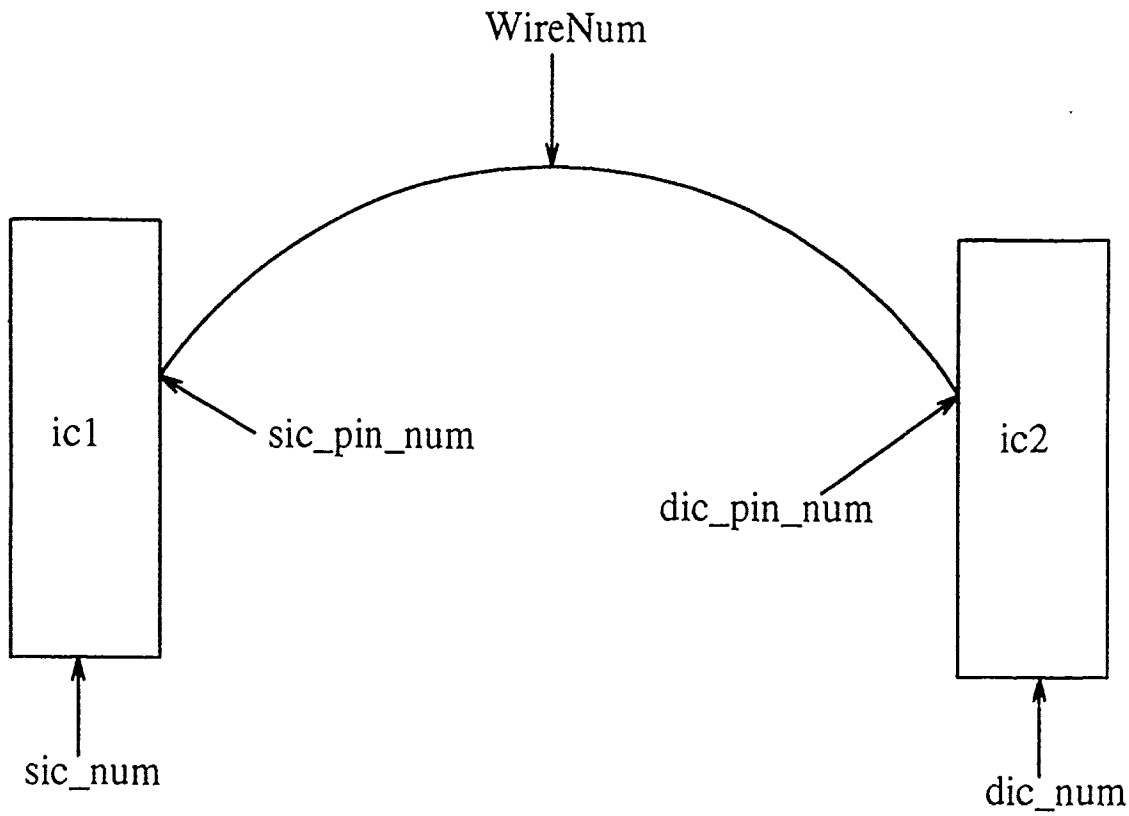


Figure 5.4. Integrated circuit connections.

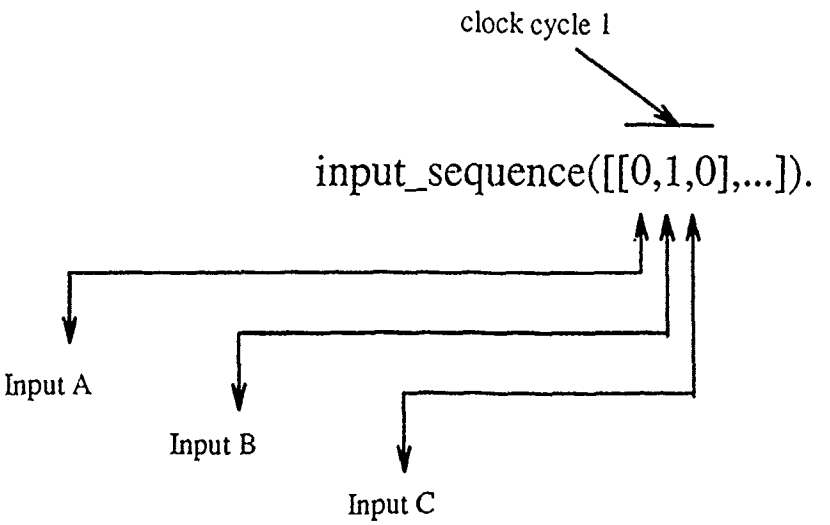


Figure 5.5. Example of an input-sequence.

if the circuit to be simulated has three inputs A, B, and C, the wire-list would have as its first input entry, a wire-name of the user's choice and the integrated circuit and pin number to which input A is to be connected. As many input A entries as desired could follow as long as the next distinctly different input entry is for input B and the associated integrated circuit to which input B is to be connected. Input B should then be followed by input C and so on. Once inputs A, B, and C have been declared, additional A, B, or C input entries can come in any order. The same sequential listing-order must be followed for the output signals (See Figure 5.5).

If the circuit design involves a connection between ground or Vcc and an input pin (for example tying the "K" input to a JK flip-flop low), then the connection must be specified. To tie an input pin to ground use a wire-list entry with the following format:

```
[0,dic_num,dic_pin_num].
```

To tie an input pin high use

```
[1,dic_num,dic_pin_num].
```

Output integrated circuit pins that are to be left floating do not need to be declared in the wire-list. TTLS will automatically ignore any floating output pins. Also, the user should not list connections to those integrated circuit pins that are normally connected to power, clock, ground, clear, or preset. TTLS will provide an accurate circuit simulation without these connections being defined.

The Integrated Circuit List. Each integrated circuit must be assigned a distinct number (called an ic-number). For example, if the circuit contains two integrated circuit packages, both might be declared as

```
ic(ic1,'7400').
ic(ic2,'7402').
```

with the ic-number being assigned at the user's discretion. If the circuit requires several integrated circuit packages of one type, a new distinct ic-number must be assigned to each package. For example, if the circuit requires six NAND-gates (SN7400 has four per package), ic1 would provide the first four and ic2 the remaining two. The two integrated circuits would be defined as

```
ic(ic1,'7400').
ic(ic2,'7400').
```

The Input-Sequence. An input-sequence is defined as a list of lists using Prolog syntax. For example, to simulate a circuit with three inputs, the input list would look like Figure 5.5. The Figure shows that for the first clock cycle, input A will be set to 0, input B will be set to 1, and input C will be set to 0. The pattern of one three element list per clock cycle can be repeated for as many clock cycles as desired. Only one input-sequence can be defined in the circuit-file. Additional input-sequences can be entered from the keyboard during the simulation. An input-sequence can be defined for an unlimited number of clock cycles regardless of whether it is listed in the circuit-file or entered from the keyboard. An example of a complete circuit-file can be found in Section 5.4.4.

5.4.3 Detailed Operational Analysis. The simulation process starts with the user typing ``[ttl]". at the Prolog prompt. The built-in interface will respond by informing the user that several files necessary to the program's execution are being loaded. The user is asked to specify the name of the input circuit-file, which may be one of the two test files provided or his own personal file. TTLS initiates a session by reading the chosen circuit-file.

Initiating the simulation causes TTLS to form the gate-list. The procedure listed in Appendix B, called `build_circuit`, does this by scanning the wire-list, ic-number list, and the TTL data file, ttldata.pro. The gate-list is formed by matching output integrated circuit connections, found either in the first two entries of a four-entry wire-list declaration or an output declaration, with any input connections found in either the last two entries of

a four-entry wire-list declaration or an input declaration. TTLS also detects any floating integrated circuit inputs while constructing the gate-list and when found, reports the ic-number and the pin number to the screen. The user must correct the wire-list before the simulation can proceed. As the final step, the gate-list is converted into Prolog executable code and asserted into the data base. A copy of the head of the circuit-rule is also asserted into the data base for later use as a generic template. Asserting the circuit-rule and corresponding head into the data base saves the computational overhead associated with remanufacturing the circuit-rule for each individual simulation cycle. Both the circuit-rule and the head will be reused an unlimited number of times. The circuit-rule is finally reported to the user, providing a gate-level description of the circuit implementation.

Following the construction of the circuit-rule, the procedure `set_up_state` instantiates each flip-flop's next-state arguments to the Boolean value 0. In order to better understand why this step is necessary, a discussion of the handling of present and next state values for memory devices is necessary. A definition of terminology is also necessary in order to avoid confusion between a simulation sequence and a simulation cycle. The former refers to the simulation of a circuit's operation over the duration of the currently specified input-value sequence; this process may require many clock cycles. The latter refers to the simulation of a circuit's operation for the duration of one entry in the input-value sequence, requiring one clock cycle.

Memory device arguments are kept as a list with the form

```
[Q,QBar,Q+,QBar+]
```

where `Q` is the present state output and `QBar` is the present state's negation, `Q+` is the future state value and `QBar+` its negation. Prior to the start of a normal simulation cycle, `Q` is instantiated to the Boolean value 0 or 1 and `QBar` is instantiated to the complement of `Q`. The variable `Q+` and its complement are not instantiated until after the simulation cycle. Once a cycle terminates, `Q` and `QBar` are no longer of interest and are discarded. `Q+` and `QBar+` now hold the desired value of present state for the next simulation cycle, hence `Q+` replaces `Q` and `QBar+` replaces `QBar`. `Q+` and `QBar+` are replaced with variables that will be instantiated during the next simulation cycle.

Prior to the start of a simulation sequence, the memory devices must be initialized to a known state. TTLS clears all memory devices before the beginning of each simulation sequence. This is performed by the `set_up_state` procedure. The procedure initializes each memory device's argument-list by instantiating each list into the following form:

```
[Q,QBar,0,1]
```

The evaluate_circuit procedure will shift the Q+ and QBar+ values into the Q and QBar positions, placing each memory device into a known starting state. Once this process is performed for each memory device in the circuit, the simulation can proceed. If the circuit's composition is purely combinational, the list of memory device arguments is empty and this preparation is not performed.

Following the preparation of the memory-device argument-list, the first simulation cycle is ready to begin. The procedure evaluate_circuit receives the input sequence value list, the memory device argument list, and a clock cycle count of 0. The memory device argument list is manipulated as discussed above by the procedure update_states. The transfer of state values is done at this level as evaluate_circuit recursively calls itself each simulation cycle until the simulation sequence is complete. A copy of the circuit-rule head is also retrieved from the data base to serve as a structural template. The retrieved head is procedurally correct with the proper functor and arity necessary to invoke the circuit-rule. However, all arguments of the copy are uninstantiated. The list of input values for the simulation cycle and the list of memory device arguments must be instantiated before the circuit-rule is invoked. This is done in one step using the built in operator univ. Univ is a built-in procedure that provides a useful way to obtain the arguments of a structure. The syntactical representation for univ is ''=..". The expression X =..L means that L is the list whose head is the functor X, and whose tail is the list of arguments of X. If a segment of code states

```
X=..[foo,A,B,C]
```

then X would be instantiated to the functor foo with arguments A,B and C.

Using univ, the uninstantiated list of input values is replaced with the head of the input_sequence list and in the same step, the list of memory device arguments is replaced with the updated version supplied by update_states. The list that holds the uninstantiated output arguments is left alone. The simulation cycle is now ready to proceed.

The fully prepared head is now sent to the procedure test_circuit as a Prolog goal. Since the head is procedurally correct, it can be used to invoke the circuit-rule. Through the process of stack instantiation, all of the uninstantiated variables in the output argument list and the list of memory device arguments are fixed. Both of these lists are returned to evaluate_circuit where the clock-cycle count is incremented, the list of output values reported to the user, and the list of memory device arguments prepared for

the next simulation cycle. This process continues until the list of input sequence values is exhausted.

The built-in TTLS user interface now asks the user if additional input sequence trials are desired. If so, the memory devices are cleared and the new input sequence is applied to the previously defined circuit-rule. If the input sequence is not provided in the proper form (if, for example, the c. :uit requires three input sequence values to be specified and the user erroneously provides only two), the head will not take on the correct format and the simulator will not be able to invoke the circuit-rule. If this happens, the interface informs the user of the problem and allows the user to either fix the error or terminate the session. A complete listing of the simulator code is given in Appendix B.

5.4.4 TTLS Combinational Circuit Example. The following code constitutes a complete circuit-file that describes a full-adder and all the code necessary to simulate its operation.

```
wire_list :- [InputA,ic1,pin2],
             [InputB,ic1,pin3],
             [InputC,ic1,pin8],
             [InputA,ic2,pin1],
             [InputB,ic2,pin2],
             [InputC,ic2,pin4],
             [W1,ic1,pin1,ic1,pin5],
             [W2,ic2,pin3,ic1,pin6],
             [W2,ic2,pin3,ic3,pin2],
             [W3,ic1,pin4,ic1,pin9],
             [W3,ic1,pin4,ic2,pin5],
             [W4,ic1,pin10,ic1,pin12],
             [W5,ic2,pin6,ic1,pin13],
             [W5,ic2,pin6,ic3,pin1],
             [Sum,ic1,pin14],
             [Carry,ic3,pin3].


ic(ic1,'7402').
ic(ic2,'7408').
ic(ic3,'7432').
```

```
input_sequence([[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],
               [1,0,1],[1,1,0],[1,1,1]]).
```

An ic-package level diagram of the circuit described by the wire-list is shown in Figure 5.6. Figure 5.7 depicts the gate-level representation of the same circuit with the interconnecting wires labeled with the wire-names from the wire-list. Figure 5.8 shows the same gate-level circuit but note that the wire-names have been replaced with the Prolog assigned variable names. Comments added to the circuit-file and the simulation session are set off with the Prolog comment syntax /*...*/.

```
Script V1.0 session started Thu Nov 14 23:19:15 1991
+-----------------------------------------+
| MS-DOS Prolog-1             Version 2.2 |
| Copyright 1983      Serial number: 0001213 |
| Expert Systems Ltd.                     |
| Oxford U.K.                             |
+-----------------------------------------+
/* starting the session.*/
?- [ttl].
Loading TTLSIM.PRO.
Loading TTLDATA.PRO.
Loading SETUP.PRO

Type 'go' at the next prompt to start the process.
Be sure to end all commands with a period.

ttl consulted.
?- go.
Two demonstration files are provided with
this simulator. The first, CKT1.PRO, describes a full-adder.
The second, CKT2.PRO,describes a pattern-detector which
generates an output (1) each time the input stream has at least
two zeros followed by an odd number of ones.
If you want to simulate your own circuit file;
type the file name at the prompt. Your file
must have the form 'filename.pro' but type only
the file name at the prompt. You may return to dos
at any time by typing 'halt.' at the prolog prompt.

Which file would you like to simulate? > (ckt1./ckt2./YourFile)
ckt1.
Loading CKT1.PRO

/* The wire-list is repeated for the convenience of the user.*/
```

```
The wire list is:
     [_117,ic1,pin2]
     [_127,ic1,pin3]
     [_137,ic1,pin8]
     [_117,ic2,pin1]
     [_127,ic2,pin2]
     [_137,ic2,pin4]
     [_177,ic1,pin1,ic1,pin5]
     [_191,ic2,pin3,ic1,pin6]
     [_191,ic2,pin3,ic3,pin2]
     [_219,ic1,pin4,ic1,pin9]
     [_219,ic1,pin4,ic2,pin5]
     [_247,ic1,pin10,ic1,pin12]
     [_261,ic2,pin6,ic1,pin13]
     [_261,ic2,pin6,ic3,pin1]
     [_289,ic1,pin14]
     [_299,ic3,pin3]
```

/* This is the gate-level circuit description which the interpreter derives from the wire-list. Note how much easier this list is to understand compared to the original wire-list. */

```
The gate level circuit is:
     nor(_117,_127,_177)
     and(_117,_127,_191)
     nor(_177,_191,_219)
     nor(_137,_219,_247)
     and(_137,_219,_261)
     nor(_247,_261,_289)
     or(_261,_191,_299)
```

/* The input and output values are printed to the screen and referenced to their particular clock cycle. */

```
CLOCK CYCLE     INPUT VALUES          OUTPUT VALUES
     1          [0,0,0]                   [0,0]
     2          [0,0,1]                   [1,0]
     3          [0,1,0]                   [1,0]
     4          [0,1,1]                   [0,1]
     5          [1,0,0]                   [1,0]
     6          [1,0,1]                   [0,1]
     7          [1,1,0]                   [0,1]
     8          [1,1,1]                   [1,1]

Do you want to input another simulation sequence? (yes./no.)>
no.
simulation over
Script completed Thu Nov 14 23:20:07 1991
```

5.4.5 TTLS Sequential Circuit Example. This example illustrates the simulation of a sequential circuit pattern-detector that when provided an input sequence of Boolean values 0 or 1, detects the occurrence of two or more 0's followed by an odd number of 1's. Derivation of the pattern-detector circuit is described in Appendix B. The complete circuit-file is

```
wire_list :-
        [InputA,ic1,pin1],
        [InputA,ic2,pin1],
        [InputA,ic3,pin9],
        [W1,ic1,pin2,ic3,pin1],
        [W1,ic1,pin2,ic2,pin4],
        [W1,ic1,pin2,ic4,pin13],
        [W2,ic3,pin3,ic4,pin2],
        [W3,ic4,pin10,ic3,pin2],
        [W3,ic4,pin10,ic2,pin2],
        [W3,ic4,pin10,ic3,pin4],
        [W4,ic4,pin6,ic2,pin5],
        [W4,ic4,pin6,ic3,pin5],
        [W5,ic2,pin3,ic4,pin3],
        [W6,ic2,pin6,ic4,pin14],
        [W7,ic3,pin6,ic3,pin10],
        [Output,ic3,pin8].
```

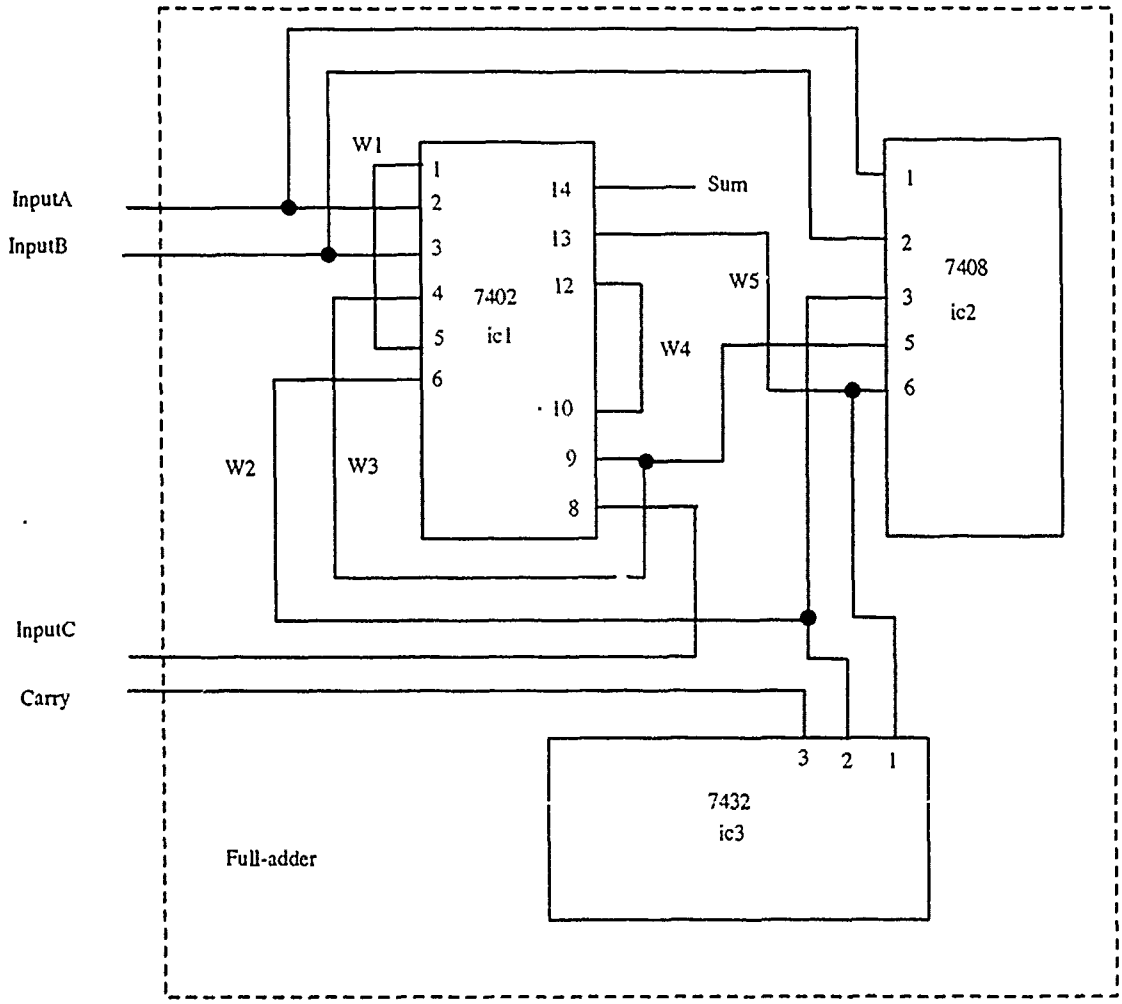


Figure 5.6. Integrated circuit diagram of a full-adder.

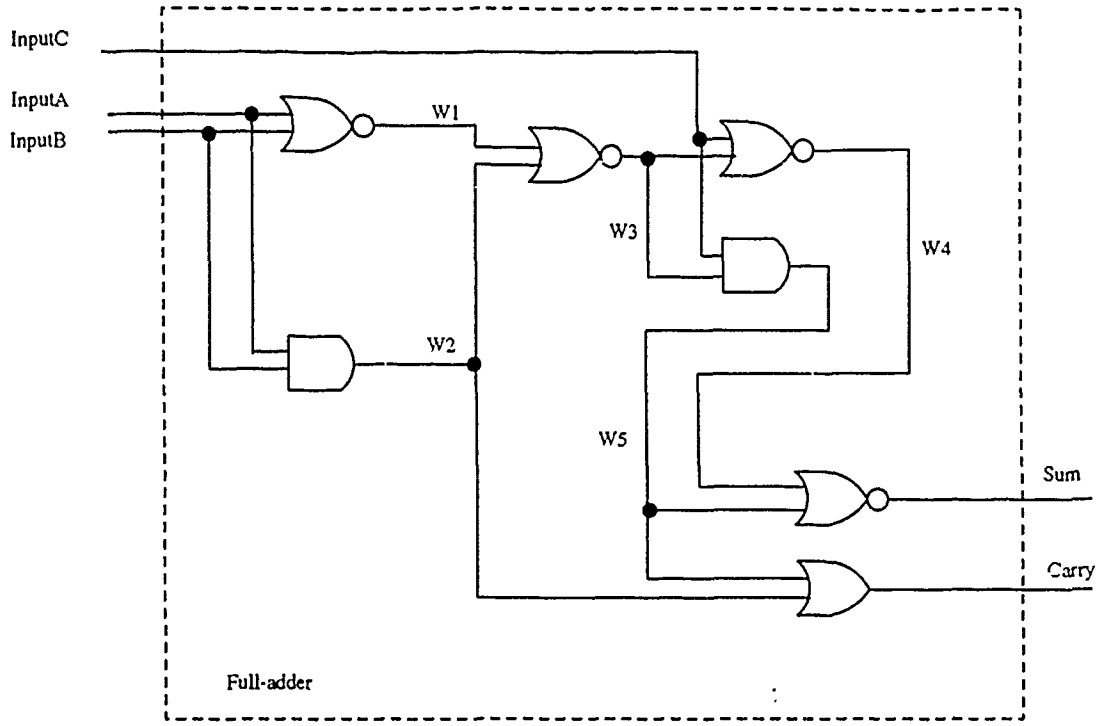


Figure 5.7. Gate-level diagram of a full-adder with wire-names.

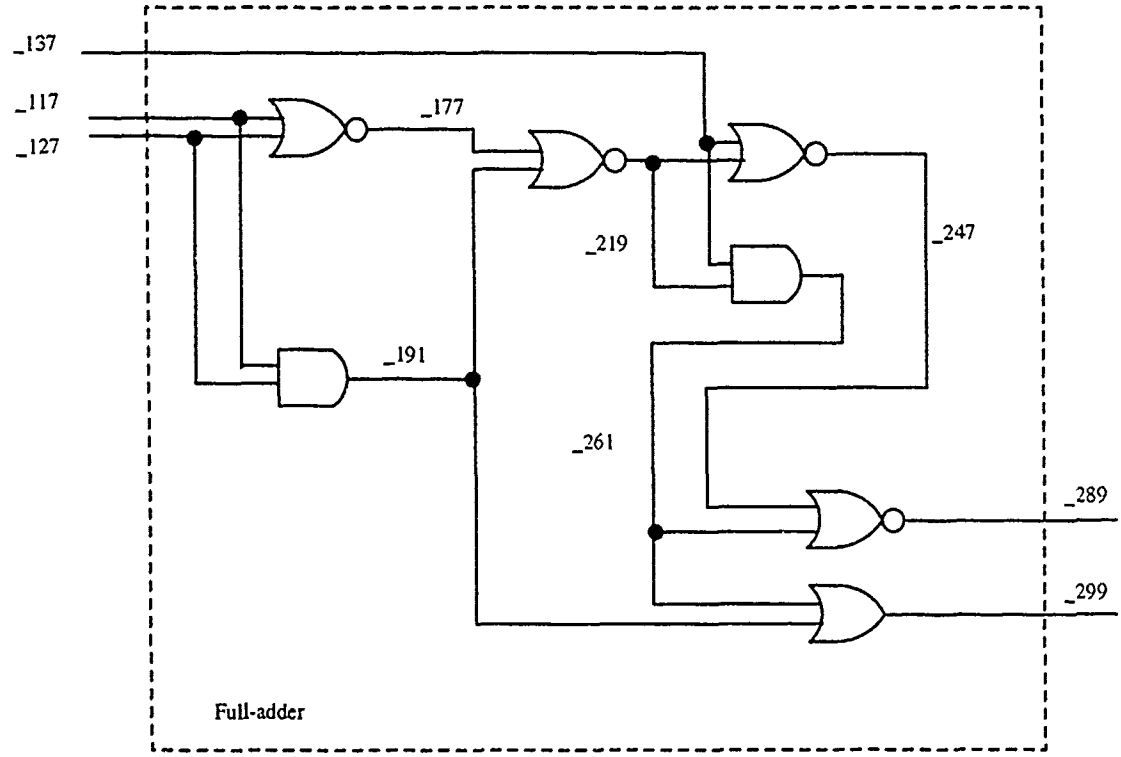


Figure 5.8. Gate-level diagram of a full-adder with variable wire-names.

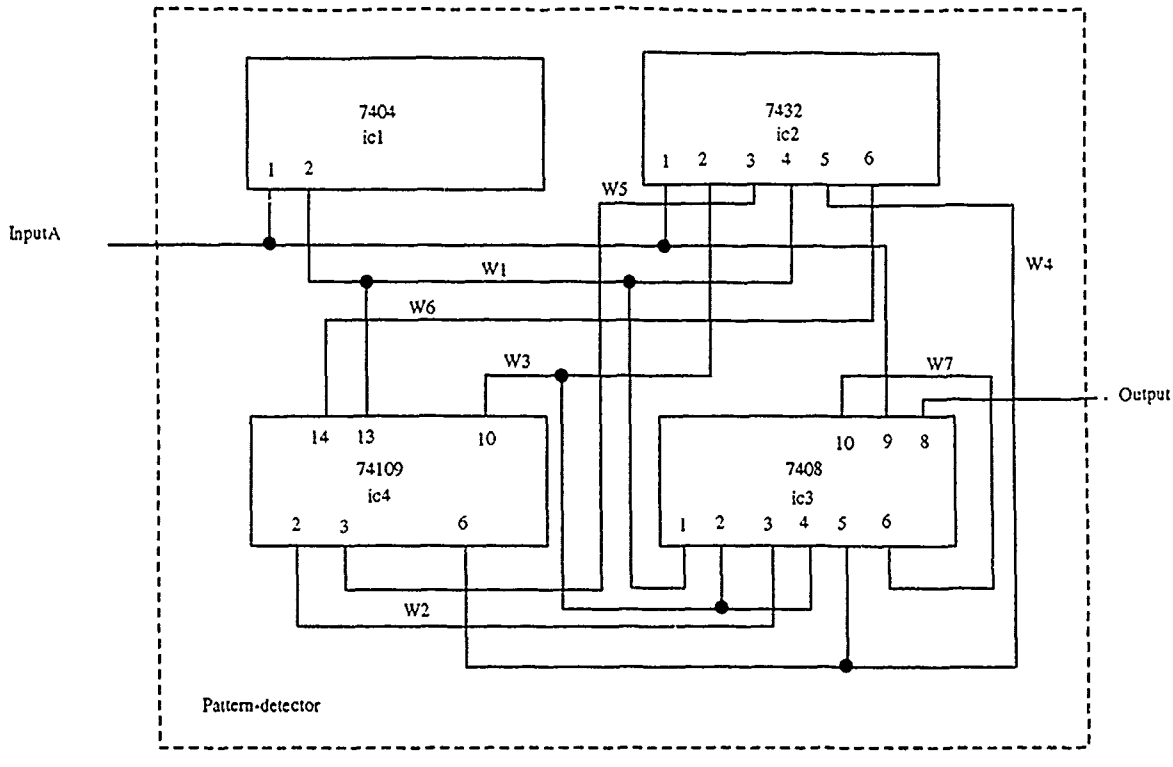


Figure 5.9. Integrated circuit diagram of a pattern-detector.

```
input_sequence([[1],[0],[0],[0],[1],[1],[1],[1]]).

ic(ic1,'7404').
ic(ic2,'7432').
ic(ic3,'7408').
ic(ic4,'74109').
```

The TTL package level diagram is shown in Figure 5.9. The gate-level circuit diagrams showing both the user provided wire names and the Prolog assigned variable wire-names are given in Figures 5.10 and 5.11.

```
Script V1.0 session started Fri Nov 15 00:53:23 1991

+-------------------------------------------+
| MS-DOS Prolog-1            Version 2.2    |
| Copyright 1983      Serial number: 0001213  |
| Expert Systems Ltd.                       |
| Oxford U.K.                               |
+-------------------------------------------+

?- [ttl].
```

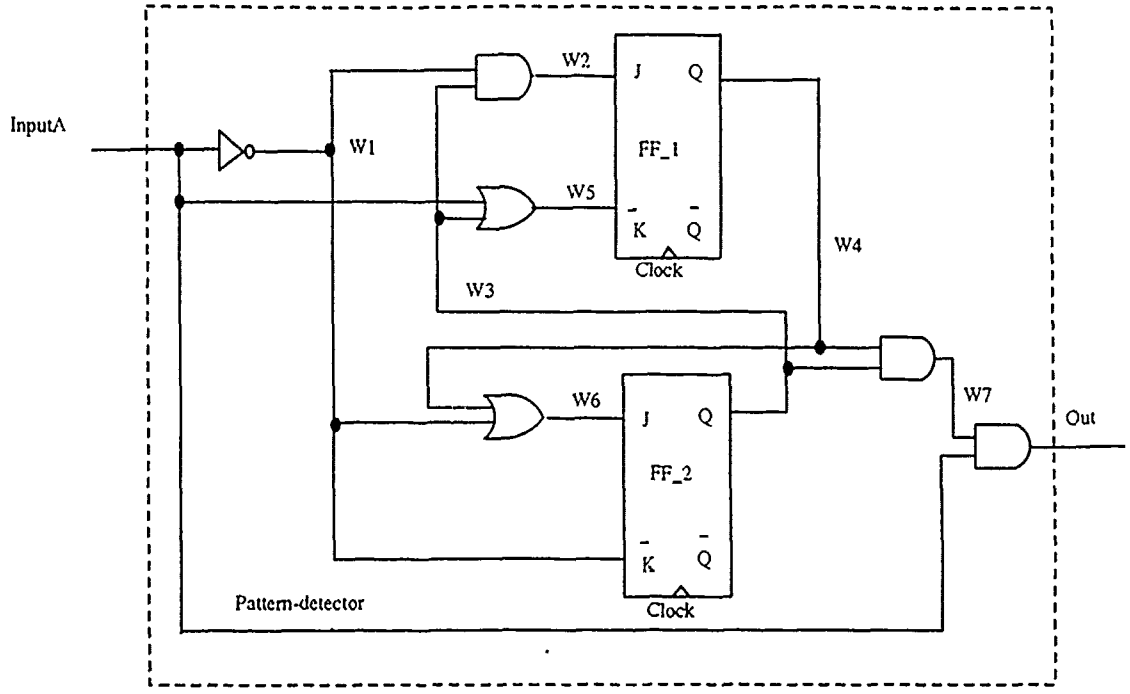


Figure 5.10. Gate-level diagram of the pattern-detector circuit with wire-names.

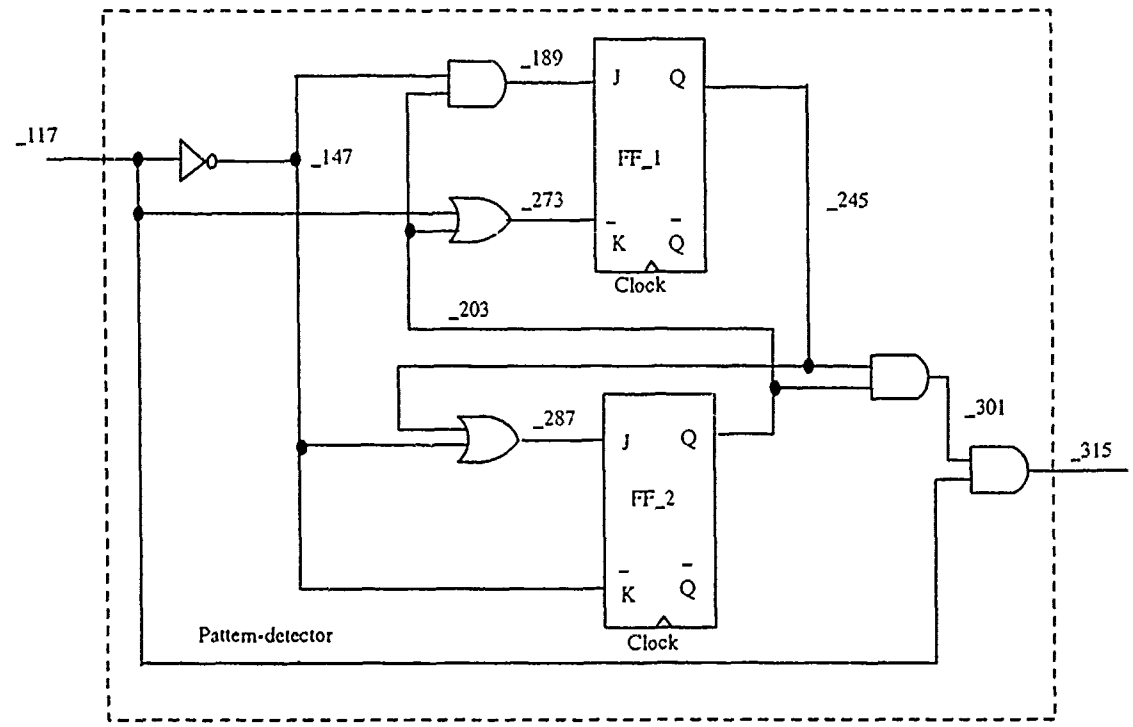


Figure 5.11. Gate-level diagram of the pattern-detector circuit with variable wire-names.

```
Loading TTLSIM.PRO.
Loading TTLDATA.PRO.
Loading SETUP.PRO

Type 'go' at the next prompt to start the process.
Be sure to end all commands with a period.

ttl consulted.
?- go.
Two demonstration files are provided with
this simulator. The first, CKT1.PRO, describes a full-adder.
The second, CKT2.PRO,describes a pattern-detector which
generates an output (1) each time the input stream has at least
two zeros followed by an odd number of ones.
If you want to simulate your own circuit file;
type the file name at the prompt. Your file
must have the form 'filename.pro' but type only
the file name at the prompt. You may return to dos
at any time by typing 'halt.' at the prolog prompt.

Which file would you like to simulate? > (ckt1./ckt2./YourFile)
ckt2.
Loading CKT2.PRO
```

/* The wire-list is printed to the screen for the user's convenience. */

```
The wire list is:
     [_117,ic1,pin1]
     [_117,ic2,pin1]
     [_117,ic3,pin9]
     [_147,ic1,pin2,ic3,pin1]
     [_147,ic1,pin2,ic2,pin4]
     [_147,ic1,pin2,ic4,pin13]
     [_189,ic3,pin3,ic4,pin2]
     [_203,ic4,pin10,ic3,pin2]
     [_203,ic4,pin10,ic2,pin2]
     [_203,ic4,pin10,ic3,pin4]
     [_245,ic4,pin6,ic2,pin5]
     [_245,ic4,pin6,ic3,pin5]
     [_273,ic2,pin3,ic4,pin3]
     [_287,ic2,pin6,ic4,pin14]
     [_301,ic3,pin6,ic3,pin10]
     [_315,ic3,pin8]
```

/* This is the gate-level circuit which the interpreter derived from the input wire-list. */

```
The gate level circuit is:
     inv(_117,_147)
     and(_147,_203,_189)
     jkbar([_287,_147],[_203,_841],[_851,_853])
     jkbar([_189,_273],[_245,_966],[_976,_978])
     or(_117,_203,_273)
     or(_147,_245,_287)
     and(_203,_245,_301)
     and(_117,_301,_315)
```

/* The sequence of input and output values are correlated to clock cycles for the convenience of the user. */

CLOCK CYCLE	INPUT VALUES	OUTPUT VALUES
1	[1]	[0]
2	[0]	[0]
3	[0]	[0]
4	[0]	[0]
5	[1]	[1]
6	[1]	[0]
7	[1]	[1]
8	[1]	[0]

/* Another test sequence of input values is entered from the keyboard. */

```
Do you want to input another simulation sequence? (yes./no.)>
yes.

Please enter the new sequence as a list of lists.
One sublist should be entered for each simulation
clock cycle desired. Each sublist should contain
the correct number of input values. Only the
boolean values 0 or 1 may be entered.
Enter your input now. Do not forget to
to follow your entry with a period.

[[1],[1],[1],[0],[0],[0],[0],[1],[1]].
```

```
CLOCK CYCLE     INPUT VALUES          OUTPUT VALUES
     1          [1]                   [0]
     2          [1]                   [0]
     3          [1]                   [0]
     4          [0]                   [0]
     5          [0]                   [0]
     6          [0]                   [0]
     7          [0]                   [0]
     8          [1]                   [1]
     9          [1]                   [0]

Do you want to input another simulation
 sequence? (yes./ no.) >yes.

Please enter the new sequence as a list of lists.
One sublist should be entered for each simulation
clock cycle desired. Each sublist should contain
the correct number of input values. Only the
boolean values 0 or 1 may be entered.
Enter your input now. Do not forget to
to follow your entry with a period.

/*Intentionally putting an incorrectly formatted input- sequence in. */
[1],[1],[1].

/*This header print-out can be fixed so that it does not show up if an erroneous input
sequence is entered. */
CLOCK CYCLE     INPUT VALUES          OUTPUT VALUES

ERROR: You did not enter your new input sequence
list in the proper format. Enter the correct form
of the input sequence.
Do you want to try again? (yes./no.)
no.
simulation over
```

5.5 The Simulation of Circuit Faults

Prolog can be used to identify faulty components by altering a circuit's operation in order to mimic improper operation. Components that are suspect need only be labeled as

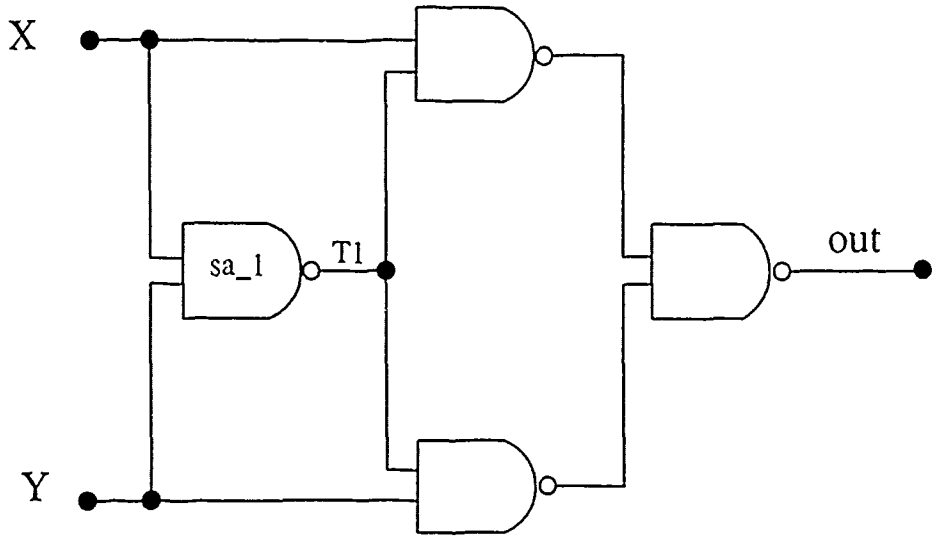


Figure 5.12. NAND-gate circuit "stuck at 1".

such and their operating characteristics altered to reflect the symptoms of the malfunction. For example, the circuit shown in Figure 5.12, when operating correctly, performs the XOR-gate function. If the circuit malfunctioned and the output was observed as that listed in Table 5.4, under the column labeled "Out", the circuit's operation could be simulated with faulty gates until the simulated operation mimics the observed faulty operation. One possible problem that would cause the symptoms could be the first NAND-gate of the circuit stuck at the logic value "1". Figure 5.12 shows the suspect gate labeled. Suppose the Prolog definition of the circuit to be altered so that the goal-entry for that particular gate is replaced by the goal

```
sa_1(X,Y,T1)
```

and the operation for the function sa_1 is defined as in Table 5.5. The simulated output of the circuit would then mimic the observed symptoms. One possibility to correct the malfunction would be to replace that NAND-gate.

This is a very simplistic example but, through the use of hierarchical descriptions, complex circuits can be troubleshot, module by module, until the faulty component is isolated.

Table 5.4. Truth-table for Figure 5.12 with the first NAND-gate "stuck at 1".

X	Y	T1	Out
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	1

Table 5.5. Truth-table for a NAND-gate "stuck at 1".

X	Y	Out
0	0	1
0	1	1
1	0	1
1	1	1

VI. Specialization

6.1 Introduction

The thrust of this chapter is to provide a Prolog implementation of the specialization process described by Clocksin [13, 14]. An extensive review of published literature, going back to the original publication of Clocksin's article in 1986, could uncover no further mention of the process. Clocksin outlined a specific example in his article on specialization [14] in order to illustrate the process, along with an algorithm that is very dependent upon his example. His description of specialization is very general in his example. Nowhere in his article does he give any details concerning implementation.

6.2 Description of the Problem

Very Large Scale Integrated (VLSI) Circuits are often designed with predefined generic circuits called standard cells. Circuits designed with standard cells tend to contain more devices than circuits that are built from basic components because of two reasons:

1. Standard cells need standard interfaces. Each connection between a cell and other cells must have some form of internal matching interface.
2. Standard cell libraries provide a manageable number of general-purpose cells. Since the standard cell designer cannot predict in advance the precise functionality required by some future designer, over-designed cells are usually produced.

After the design of a VLSI circuit has been completed, it is desirable to remove any extraneous circuitry. The Prolog code described in this chapter will remove all circuitry that does not directly contribute to the final output of the circuit. Redundant components are removed by tracing through the circuit's hierarchy and identifying any components that have unused outputs. A component that has all of its outputs unused can be removed with impunity. This removal may lead to the removal of other components within the circuit. This entire process has been named *specialization* by Clocksin.

An implementation of Clocksin's process of specialization is given in this chapter. The key to understanding this implementation lies in understanding the recursive properties of Prolog. As the interpreter proceeds through the levels of recursion, its inherent search strategy enables it automatically to keep track of where it has been. This allows the various levels of abstraction to be properly disassembled and reassembled as the interpreter

moves through a circuit's hierarchy. The programmer needs only to define the upper level of a circuit's nodal structure and provide the constituent lower-level generic module templates. The interpreter will make the appropriate unifications and instantiations during the execution of the program.

Inverters, NAND, AND, XOR, and OR-gates are considered to be primitive throughout this chapter. The investigation into Clocksin's process discussed in this chapter will be confined to combinational circuits.

6.3 Clocksin's Specialization Example

The circuit Clocksin specialized is shown in Figure 6.1. By convention, the circuit to be specialized must have one or more used input nodes and one or more used output nodes. Only the submodules of the circuit to be specialized are allowed to have entirely unused sets of output nodes. Note that one of the output nodes, Co, specified as part of the submodule M2, is unused. Given the top-level goal of specializing the two-bit adder-subtractor, defined as

```
twobit(a1,b1,a2,b2,c,as,s1,s2)
```

the program proceeds as follows:

1. Inspect the modules of the two-bit adder-subtractor, finding the one-bit adder-subtractor M2 (see Figure 6.1) with an unused output node, Co. Recursively enter the definition of M2 with the goal of further investigating it.
2. Tracing the unused output node, Co, of M2 leads to the OR-gate, G1, as shown in Figure 6.2. Since M2's output is not used elsewhere and an OR-gate is defined as a primitive gate, it can be removed. Removing OR-gate G1 leaves the AND-gate G2 and the XOR-gate G3 with unused output nodes that can also be removed as neither of their outputs are used elsewhere. The half-adder M4 (see Figure 6.3) now has an unused output node T1; hence it is entered recursively with the goal of specializing it. Note that the internal connection, T1, is hidden from the definition of `twobit`.
3. Two gates can be removed from the half-adder M4. Both the Inverter G4 and the NAND-gate G5 are not needed (see Figure 6.3). No other gates or modules with unused outputs remain in the half-adder M4 so a unique identifier for the specialized half-adder is generated and M4's refined definition is entered into the database. The new module replaces the half-adder M4 (see Figure 6.4). Control then returns to the next higher level in the circuit hierarchy.

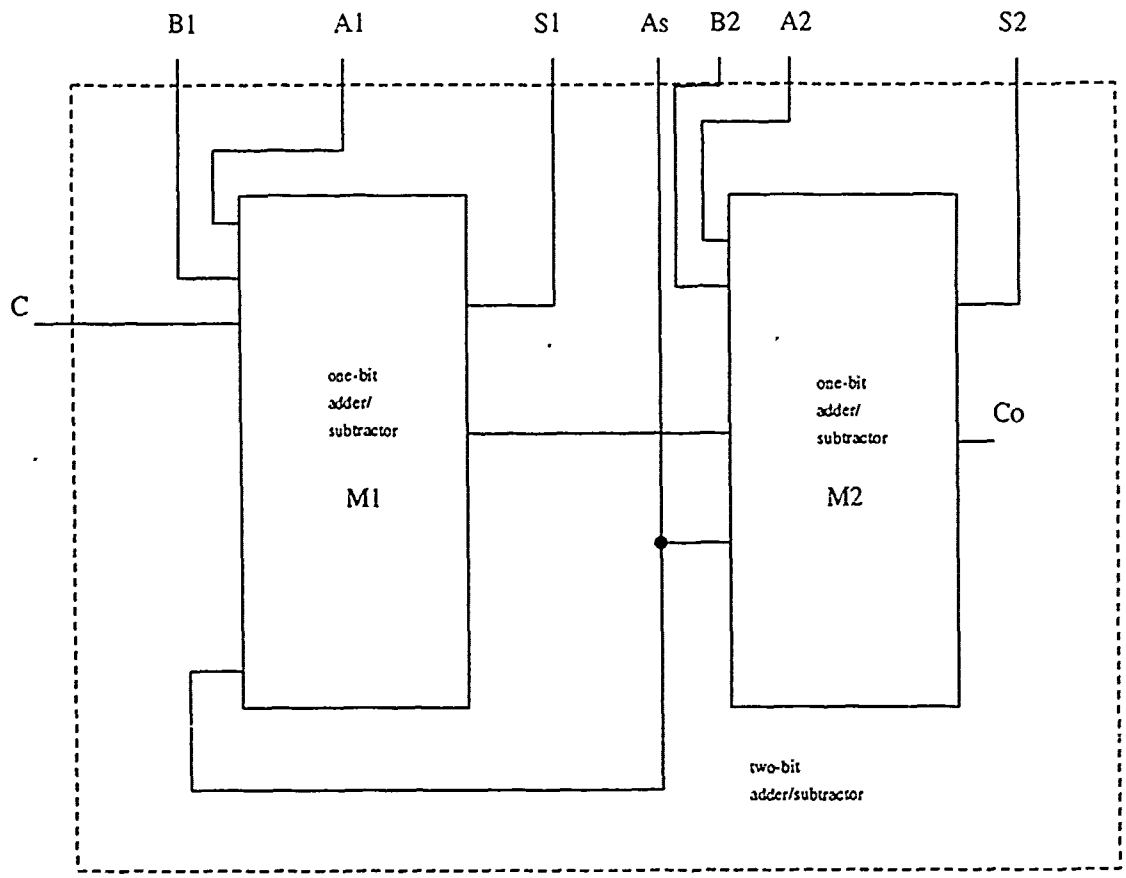


Figure 6.1. Circuit schematic for a two-bit adder-subtractor.

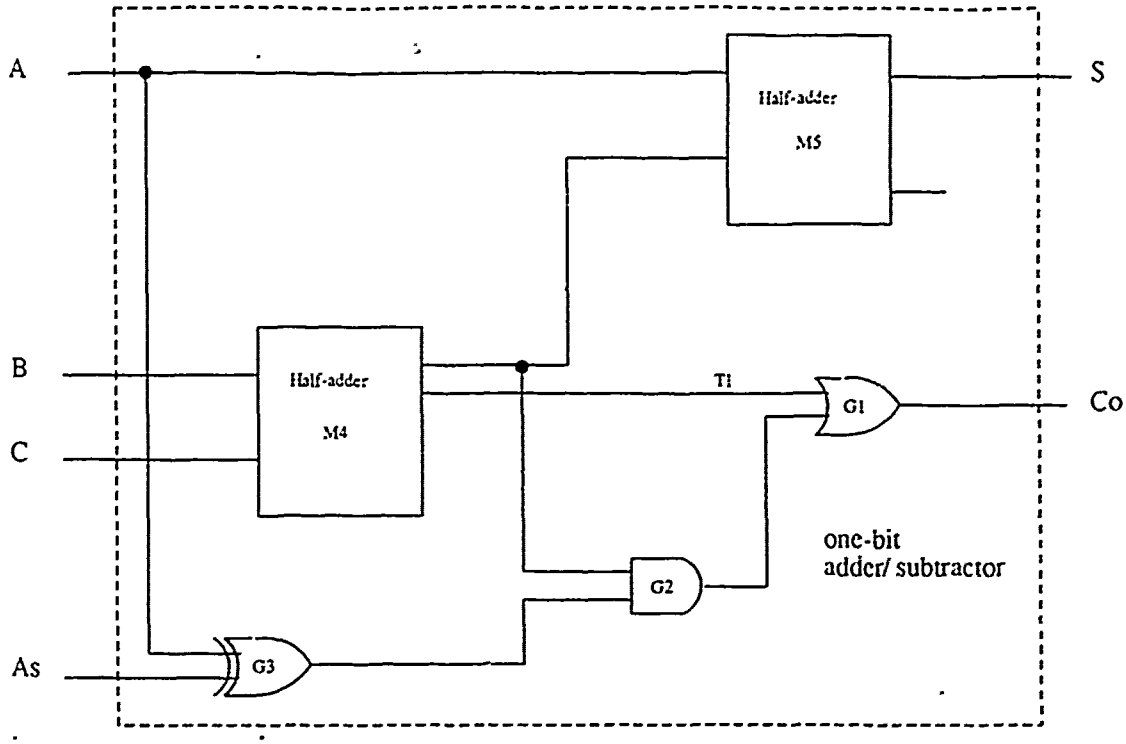


Figure 6.2. Circuit schematic for a one-bit adder-subtractor.

4. The remaining half-adder, M5, of the one-bit adder-subtractor, M2, also has an unused output node as shown in Figure 6.2. Because of this, the definition of module M5 is now entered with the goal of specializing it. As was previously done for the half-adder module M4, the module M5 is reduced to the circuit of Figure 6.4 and a unique identifier generated for the new version. The new version (Figure 6.4) is also added to the database and control now returns back to the next higher level of the circuit hierarchy, that of the one-bit adder-subtractor. All possible specializations have been carried out at this level; therefore control returns to the two-bit adder-subtractor level.

5. The two-bit adder-subtractor module M2 is replaced by its specialized version (Figure 6.5) and the search terminates because all unused outputs at every level have been investigated. The final hierarchical representation of the circuit is shown in Figure 6.6.

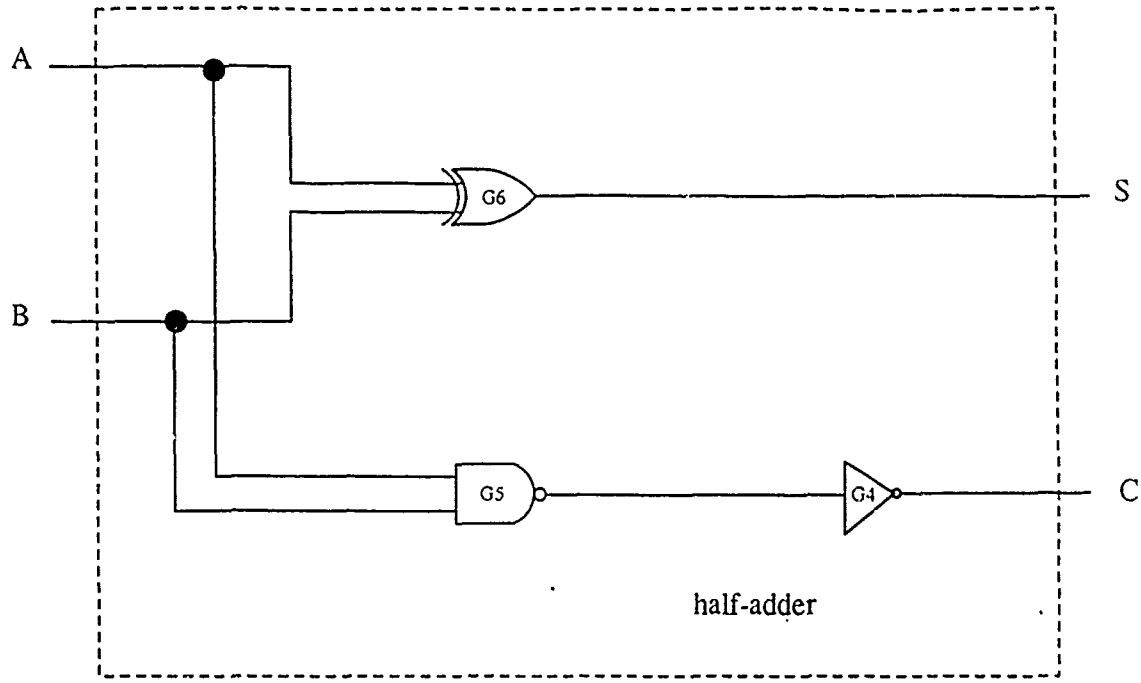


Figure 6.3. Circuit schematic for a half-adder.

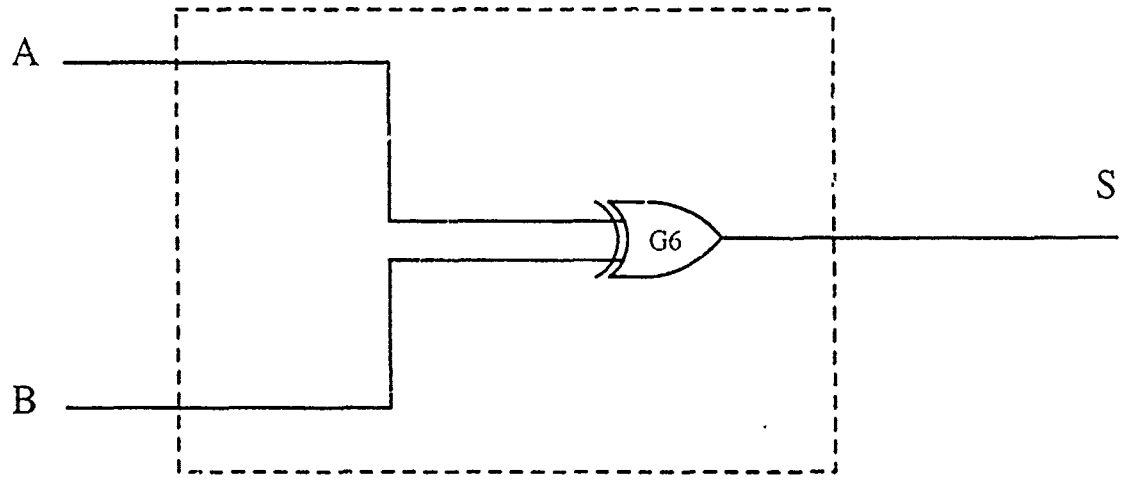


Figure 6.4. Specialized version of a half-adder.

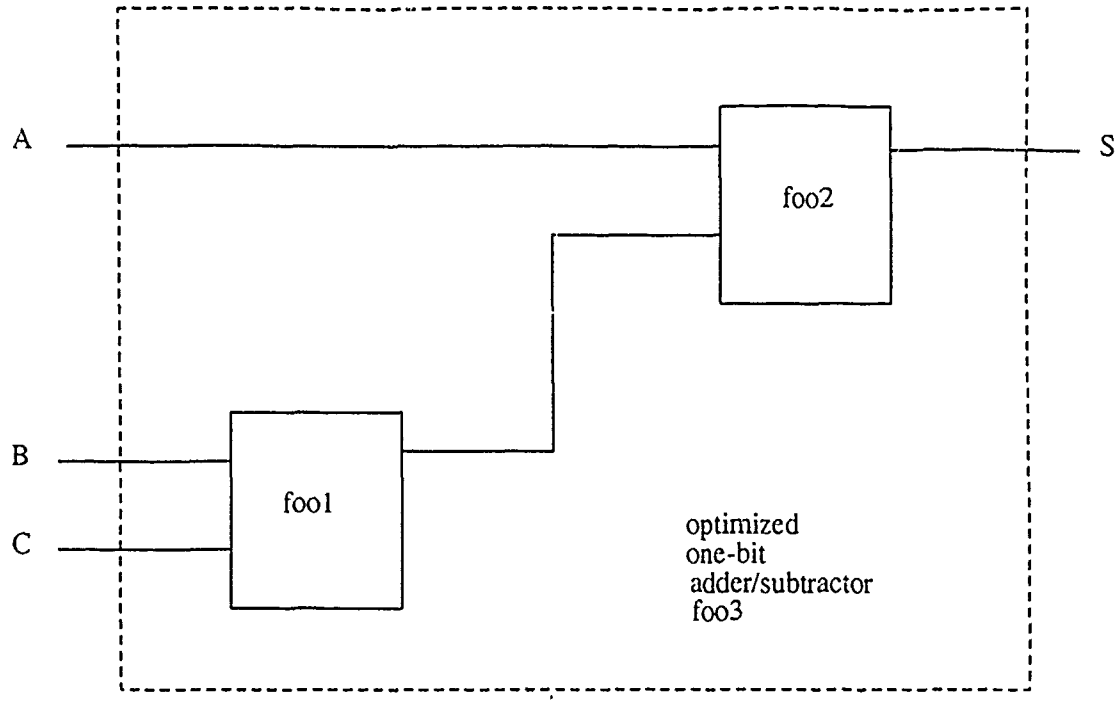


Figure 6.5. Specialized version of the one-bit adder-subtractors.

6.4 The Development of the Specialization Code

This chapter introduces a procedure developed for the specialization of a more general class of circuits than that represented by Clocksin's example. A more general approach must be capable of handling four types of situations relating to specialization:

1. Primitive modules that have at least one used output node cannot be specialized.
2. If all the output-nodes of a higher-level module are used, then that module cannot be specialized.
3. If all output-nodes of a primitive module are unused, then that module should be removed from the circuit. This removal may result in disconnecting the output nodes of other primitive or higher-level modules. These modules may also be able to be removed.
4. If a higher-level module has some, but not all of its output-nodes unused, it is necessary to recursively descend the modules hierarchy to determine if any submodules can be removed.

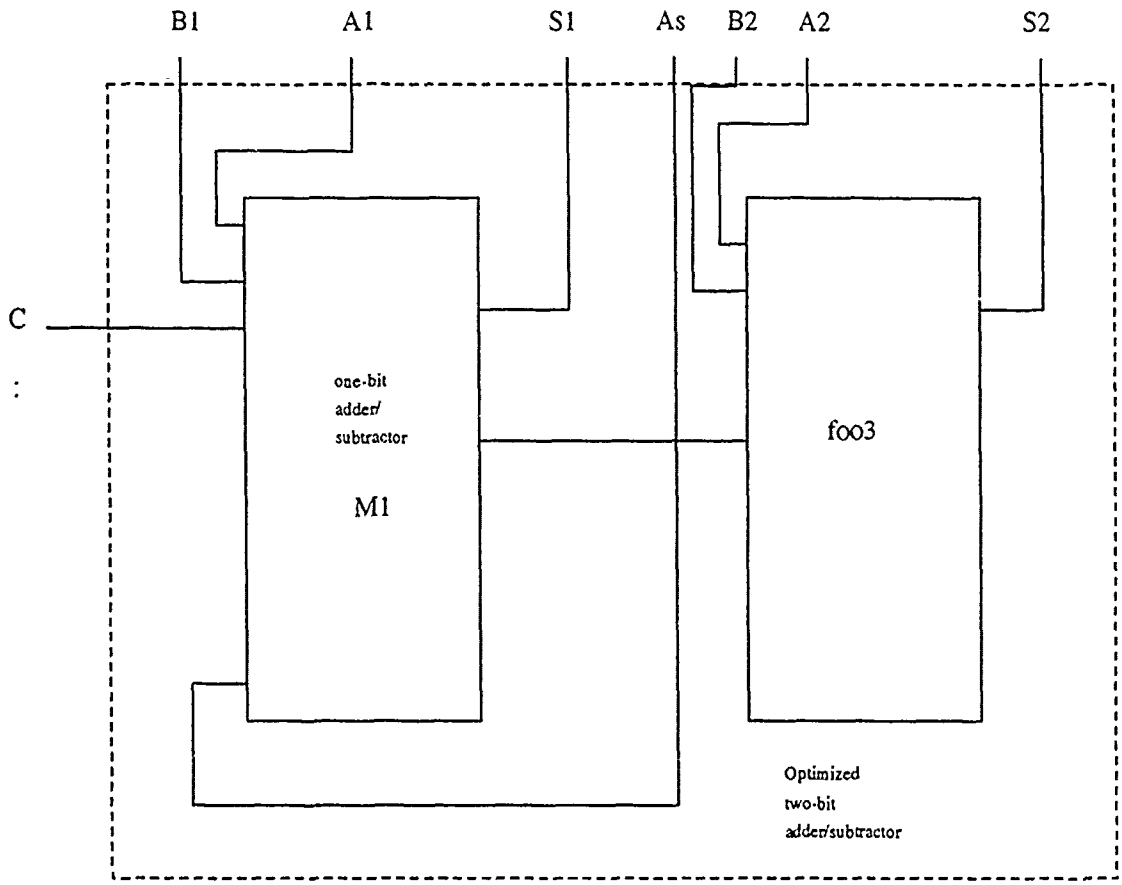


Figure 6.6. Specialized version of the two-bit adder-subtractor.

The Code. In order for Clocksin's specialization process to work, the user must specify a circuit definition at the highest level, any submodule definitions, the direction of submodule ports, and gate status. Gate status is declared explicitly with the primitive predicate explained earlier in Chapter 4. The direction of submodule ports is declared with a generic predicate. The general form of the declaration is

```
direction(GateType,InputPorts,OutputPorts).
```

For example, the fact `direction(nand(A,B,C),[A,B],[C])` declares that any NAND-gate fact of arity three that can be found has three ports. The first two arguments define input ports and the third argument defines an output port.

The direction of the ports must be declared for any submodule but not for the upper level definition of the circuit to be specialized. Consider the circuit defined in Clocksin's specialization example [13]. The circuit definition is

```
twobit(A1,B2,A1,B2,C,As,S1,S2) ;-
   addsub(A1,B1,C,As,S1,T),
   addsub(A2,B2,T,As,S2,Z).
```

The upper-level circuit, `twobit`, does not need a fact that describes the direction of its ports. The direction of twobit's ports is determined automatically during the execution of the process that specializes `twobit`. The direction of `twobit` is asserted into the database for future use. This is an additional feature that Clocksin mentioned but said he did not provide in his implementation of the process. Submodules under `twobit` must have their port directions declared. The specialization of `twobit` requires direction declarations for `addsub` which, in turn, requires declarations of direction for its submodules and so on until the primitive level of the hierarchy is reached. A complete listing of the definition of `twobit` and the ancillary procedures required to specialize `twobit` can be found in Appendix C. Section C.1.5, under the code for `test4`.

Appendix C also contains test examples besides the one for `twobit`. Four test cases are considered, including `twobit`, in order to illustrate the specialization process as it operates at the various levels of circuit hierarchy. The purpose of each test case is documented. "Before" and "after" schematic representations are provided along with verbose simulation sessions that allow the execution of the process to be followed in detail.

The execution of the specialization code given in Appendix C, regardless of the circuit to be specialized, starts with an invoking call to the procedure `scan(Head)`. The variable

Head is the definition of the circuit to be specialized. No variables can appear in the circuit definition at this level. For example, the two-bit adder-subtractor of Figure 6.1 is described as `two-bit(a1,b1,a2,b2,c,as,s1,s2)`. No unused outputs are allowed to appear in the circuit definition at this level; therefore if there are any any unused outputs, they must appear at some lower level in the hierarchy. The top level circuit definition is broken down into a head predicate and the body goals. Each goal of the body represents one module, that can be either primitive or higher-level. The head arguments are checked against the goal arguments in order to detect any unused output nodes. A list of the goals and any unmatched nodes are passed to the `analyze(Goals,Acc,Circuit)` procedure. The procedure `analyze` examines each of the goals and separates those goals with unused output nodes from those with no unused output nodes. If a goal does not have any unused output nodes, it is stored in the accumulator (Acc) for future use. If there are no unused output nodes found after examining all the goals, the original circuit definition is returned to the scan procedure via the variable `Circuit` and the process terminates without changing the original circuit definition.

If a goal is found that has at least one unused output, the goal is sent to another procedure called `specialize`. If the goal is primitive, it is removed from the circuit's definition and the list of unused nodes is updated accordingly. When a module is identified as being removable, its input nodes must be tested to determine if they can be considered extraneous to the circuit. A simple test is used to determine if a module's input nodes can be added to the list of unmatched nodes once that module has been removed. Basically, if there is no other requirement for the input node designator throughout the circuit at any level, then that input node can be added to the list of unmatched nodes.

If `specialize` receives a higher-level module from `analyze`, then the definition of the module is recursively entered by resubmitting it to `analyze`, receiving the specialized version back, updating the list of unmatched nodes, and returning control back to the original invoking call. `Analyze` re-examines the accumulator each time a module is specialized to ensure that all possible redundant connections are identified. Once the accumulator contains only those modules that cannot be further reduced, a new unique identifier is generated and asserted into the database. The direction of the new module is also determined and asserted into the database. Although there is no direct use for this feature currently, future implementations or applications of this procedure may find it convenient.

Control then returns to the next level up until control finally reverts to the topmost level where the results of the process are displayed on the screen. The final circuit is

expressed as a hierarchy of modified modules. The composition of any module can be found if the built-in procedure `listing(X)` (where `X` is the functor of the desired module) is used.

6.4.1 A Specialization Session. Appendix C contains the specialization code (`speccode.pro`) and four verbose test sessions. The sessions are verbose so the reader can easily follow the execution of the program. The last test, `test4` is repeated here in nonverbose form. Comments, that were added to improve the clarity of the session, are set-off with the standard Prolog1 syntax for comments `/*...*/`.

```
/*test4 circuit definition.*/

     test4 :-
       scan(twobit(a1,b1,a2,b2,c,as,s1,s2)).


     twobit(A1,B1,A2,B2,C,As,S1,S2) :-
       addsub(A1,B1,C,As,S1,T),
       addsub(A2,B2,T,As,S2,Unused).
```

/*The following submodule definitions are necessary to carry out test4. Section C.1.5 further explains the following procedures. **/

```
     addsub(A,B,C,As,S,Co) :-
       halfadd(B,C,T1,T2),
       halfadd(A,T1,S,Unused),
       xor_gate(A,As,T3),
       and_gate(T1,T3,T4),
       or_gate(T2,T4,Co).


     halfadd(A,B,S,C) :-
       xor_gate(A,B,S),
       nand_gate(A,B,T),
       nnot_gate(T,C).
```

```
direction(addsub(A,B,C,As,S,Co),[A,B,C,As],[S,Co]).
direction(halfadd(A,B,S,C),[A,B],[S,C]).
direction(or_gate(A,B,Out),[A,B],[Out]).
direction(xor_gate(A,B,Out),[A,B],[Out]).
direction(and_gate(A,B,Out),[A,B],[Out]).
direction(nnot_gate(A,B),[A],[B]).
direction(nand_gate(A,B,C),[A,B],[C]).

primitive(or_gate).
primitive(xor_gate).
primitive(and_gate).
primitive(nnot_gate).
primitive(nand_gate).
```

/*Test Results.*/

```
Script V1.0 session started Mon Oct 14 15:04:07 1991

Microsoft(R) MS-DOS(R) Version 4.01
             (C)Copyright Microsoft Corp 1981-1988

A:\THESIS\CODE>prolog


+-------------------------------------------+
| MS-DOS Prolog-1              Version 2.2  |
| Copyright 1983      Serial number: 0001213 |
| Expert Systems Ltd.                       |
| Oxford U.K.                               |
+-------------------------------------------+
```

/* Loading the specialization code-file. */

```
?- [speccode].
speccode consulted
?- test4.
```

/* This is the modified circuit definition generated by the process. Notice that the circuit is structured hierarchically. The built-in procedure "listing" is used to expose each level of the hierarchy. */

```
asserting foo4(a1,b1,c,as,a2,b2,s2,s1):-[foo3(a2,b2,T2,s2),
addsub(a1,b1,c,as,s1,T2)]
```

```
retracting [a2,b2,T2,as,a1,b1,c,as]

asserting direction(foo4(a1,b1,c,as,a2,b2,s2,s1),
[a1,b1,c,as,a2,b2],[s2,s1]))

Circuit =foo4(a1,b1,c,as,a2,b2,s2,s1)
yes
?- listing(foo3).
foo3(a2,b2,'T2',s2) :-
        [foo2(b2,'T2','T6'),foo1(a2,'T6',s2)] .


yes
?- listing(foo2).
foo2(b2,'T2','T6') :-
        [xor_gate(b2,'T2','T6')] .


yes
?- listing(foo1).
foo1(a2,'T6',s2) :-
        [xor_gate(a2,'T6',s2)] .


yes
?- halt.

A:\THESIS\CODE>exit
Script completed Mon Oct 14 15:05:44 1991
```

VII. Results and Recommendations

7.1 Extraction

7.1.1 Results. The basic principles of circuit extraction were explained in Chapter 4. The extraction process is a fairly simple process if the following assumptions are made.

1. Extractions will only be done within the current level of the hierarchy. If some of the constituent components to be extracted lie at one level of the hierarchy and the rest lie at another level, to remove the group and reassert some higher-level-level structure could completely change the overall function of the circuit. Prolog programs that ensure that the extraction process will not violate connectivity are being developed [26:4-8.1].
2. Extractions can be done without consideration for critical-path delays. If timing is a crucial factor in the design of a circuit, extracting groups of components with some set propagation delay time and inserting a substitute component with a different delay time could seriously affect the circuit's operation. Extraction programs that account for changes in critical path delay are being developed at AFIT [25].

7.1.2 Recommendations. The extraction process can be used in many areas of digital circuit design as long as the process is closely monitored by skilled engineers. More work could be done to shift the burden of verifying the correctness of the extraction process from the engineer. This will involve developing sophisticated extraction programs that exploit advanced logic programming techniques. Work is proceeding at AFIT on advanced applications of logic-programming to circuit extraction [25]

7.2 Simulation

7.2.1 Results. The simulation of digital circuit operation can be quick and efficient when Prolog is used. TTLS provides a convenient method for the testing of digital circuits on the TTL level and could easily be modified to simulate circuits designed with lower-level components. Several digital circuit simulators are used at AFIT, but they all operate much slower than TTLS. TTLS achieves its high simulation speed through the use of "structured-invocation" and "stack instantiation" (explained in detail in Chapter 5). TTLS's utilization of structured invocation and stack instantiation distinguish TTLS from other simulators

developed in the past. An extensive review of the literature did not uncover another simulator that employed these qualities.

7.2.2 Recommendations. TTLS can be developed into a large-scale digital circuit simulator capable of providing very fast simulation results over a broad range of TTL device types. The development of a large scale Prolog-based simulator could be used by the VLSI laboratory as a simulation tool and by the electronics laboratory as a teaching aid. The concept of "machine intelligence" could also be embodied in a TTL simulator by incorporating some of the principles associated with extraction into the simulator's design. For example. The simulator could scan the wire-list for groups of gates that perform a specified function (similar to the XOR-gate function performed by four NAND-gates discussed in Chapter 4) and suggest that the four NAND-gates be replaced with one XOR-gate. This would be a very attractive option if XOR-gates were used elsewhere in the test circuit.

TTLS also has the potential to be developed into an effective fault diagnosis tool. Given that a device or a number of devices in a circuit were faulty, TTLS could be used to simulate the defective circuit's operation and detect devices or combinations of devices capable of causing the malfunction. Specific heuristics could be developed to guide TTLS in its search for defective components. Because of the completeness of Prolog's built-in search strategy, all possible combinations of defective devices capable of causing the malfunction would be found.

7.3 Specialization

7.3.1 Results. The specialization of a general class of electronic circuits was shown to be practical. Critical path timing constraints and hierarchical boundaries are not a problem in the specialization process as only those components that do not directly contribute to the circuit's operation are removed. The code developed for this thesis has been tested over a range of circuit designs. Each test removed all redundant components and correctly derived the new circuit specification. The original algorithm proposed by Clocksin was expanded upon by including another predicate, `direction`, in the scheme. When a specialized submodule was inspected by the user, it was not easy to distinguish input ports from output ports. However, by storing the input/output status of a specialized module's ports in the form of the `direction` predicate, the status could easily be determined. Prior to the development of the specialization code discussed in Chapter 6.

an extensive literature review uncovered several journal articles reporting the development some of the concepts proposed by Clocksin in his original article [12], but no mention of any further development of his specialization process was found.

7.3.2 Recommendations. The VLSI program at AFIT does not currently possess a design tool capable of removing redundant circuitry from gate-level designs and rewriting the circuit's nodal connections to reflect the change. The concept of specialization and the code developed for this thesis should be integrated into the VLSI program.

Specialization represents only one member of the class of optimization procedures that could be developed. The specialization code can be enhanced by developing other programs to perform circuit transformations based on some heuristic other than the minimization of a circuit's gate count. For example, a program could be developed that would perform critical path timing analysis and transform a circuit from a slow or regular-speed version into a high-speed equivalent. Several different programs could also be developed to handle the different aspects of circuit synthesis based on logic programming techniques. Circuit synthesis systems such as BC [32], and Socrates [22, 29, 21, 30] were developed several years ago and do perform circuit synthesis, but the bulk of the process is performed by an expert system usually written in a language other than Prolog. Developing a logic circuit synthesis system based solely on logic programming would be a large task, but one worthy to undertake.

Appendix A. *Extraction Code and Test Results*

A.1 Extraction Code

A.1.1 Prolog Code for the Extraction Process

```
/*************************************************************/
/*                    extract.pro                            */
/*  This procedure will extract all possible Full Adder      */
/*  circuits from any input netlist.  The procedure will     */
/*  extract all NAND-gate configurations of XOR-gates        */
/*  first. Once all XOR-gates have been found, full adder    */
/*  combinations are sought. For simplicity, only NAND       */
/*  and XOR-gates are used. The built-in procedure           */
/*  "listing(X)" is used to print database entries for       */
/*  the variable "X" to the screen.                          */
/*                                                           */
/*  A typical session would be:                              */
/*     ?- consult('extract.pro').                            */
/*     yes                                                   */
/*     ?- consult('netlist.pro').                            */
/*     yes                                                   */
/*     ?- extract_full_adders.                               */
/*     yes                                                   */
/*     listing(full_adder).                                  */
/*     full_adder(x,y,z,s,c).                                */
/*     yes                                                   */
/*************************************************************/
/*******************code starts here**************************/

extract_full_adders :-
   find_xor_gates,
   find_full_adders.


/*  The input nodes could be transposed. To compensate,      */
```

```
/*  the code is written so that inputs X and Y are the same */
/*  as inputs Y and X. The built-in procedure "fail" is     */
/*  used to force the interpreter to find all possible      */
/*  solutions. "retract" and "assert" are not undone        */
/*  during the backtracking caused by the "fail" procedure. */


find_xor_gates :-
  (nand_gate(X,Y,T1);
   nand_gate(Y,X,T1)),
  (nand_gate(T1,Y,T3);
   nand_gate(Y,T1,T3)),
  (nand_gate(X,T1,T2);
   nand_gate(T1,X,T2)),
  (nand_gate(T2,T3,Out);
   nand_gate(T3,T2,Out)),
  (retract(nand_gate(X,Y,T1));
   retract(nand_gate(Y,X,T1))),
  (retract(nand_gate(T1,Y,T3));
   retract(nand_gate(Y,T1,T3))),
  (retract(nand_gate(X,T1,T2));
   retract(nand_gate(T1,X,T2))),
  (retract(nand_gate(T2,T3,Out));
   retract(nand_gate(T3,T2,Out))),
  assert(xor_gate(X,Y,Out)),
  fail.

find_xor_gates.


/* The search is the same as for the XOR-gates, only the       */
/* pattern (what we are looking for) changes.                  */


find_full_adders :-
```

```
    (xor_gate(X,Y,T1);
     xor_gate(Y,X,T1)),
    (xor_gate(Z,T1,S);
     xor_gate(T1,Z,S)),
    (nand_gate(X,Y,T2);
     nand_gate(Y,X,T2)),
    (nand_gate(Z,T1,T3);
     nand_gate(T1,Z,T3)),
    (nand_gate(T2,T3,C);
     nand_gate(T3,T2,C)),
    (retract(xor_gate(X,Y,T1));
     retract(xor_gate(Y,X,T1))),
    (retract(xor_gate(Z,T1,S));
     retract(xor_gate(T1,Z,S))),
    (retract(nand_gate(X,Y,T2));
     retract(nand_gate(Y,X,T2))),
    (retract(nand_gate(Z,T1,T3));
     retract(nand_gate(T1,Z,T3))),
    (retract(nand_gate(T2,T3,C));
     retract(nand_gate(T3,T2,C))),
    assert(full_adder(X,Y,Z,S,C)),
    fail.

find_full_adders.
```

A.2 Netlist Code

A.2.1 Prolog Code for the Netlist

```
/*********************************************************/
/*                    netlist.pro                        */
/*  This code descibes the test circuit used with the    */
/*  procedure "extract.pro" for the full adder extraction */
/*  example. Exactly one full-adder is modeled.          */
/*********************************************************/

/* First XOR-gate */

nand_gate(x,y,t1).  /* note that internal connections are  */
nand_gate(y,t1,t3). /* modelled with the letter "t"        */
nand_gate(x,t1,t2). /* followed by some number.            */
nand_gate(t2,t3,t4).

/* second XOR-gate  */

nand_gate(t4,z,t1).
nand_gate(t4,t1,t2).
nand_gate(z,t1,t3).
nand_gate(t2,t3,s).


/* remainder of the gates in the full adder */

nand_gate(x,y,t6).
nand_gate(z,t4,t5).
nand_gate(t6,t5,c).
```

A.3 Extraction Test Results

```
/*********************************************************/
/*                    TEST RESULTS                       */
/* The program "extract.pro" and "netlist.pro" are needed */
/* to produce the test results given below. "extract.pro" */
/* provides the code to perform the extract function.     */
/* "netlist.pro" is a netlist of NAND-gates which describe */
/* a full-adder. "extract.pro" will search through any    */
/* netlist given to it, and extract all the NAND-gate     */
/* full-adder circuits which it can find. In this example, */
/* "extract.pro" extracts all the full-adders from        */
/* "netlist.pro". When a full-adder is found, the         */
/* constituent NAND-gates are retracted from the database */
/* and a full-adder is asserted in their place.           */
/*********************************************************/
Script V1.0 session started Mon Sep 02 12:44:22 1991

Microsoft(R) MS-DOS(R) Version 4.01
            (C)Copyright Microsoft Corp 1981-1988

J:\THESIS\CODE>prolog

+-----------------------------------------+
| MS-DOS Prolog-1             Version 2.2 |
| Copyright 1983      Serial number: 0001213 |
| Expert Systems Ltd.                     |
| Oxford U.K.                             |
+-----------------------------------------+

?- [extract].   /* This command loads "extract.pro". */

extract consulted.


/* The built-in predicate "listing(X)" will find all      */
```

```
/* expressions for X and print them on the screen.          */

?- listing(nand_gate).  /* Checking the netlist to ensure */
                        /* that it was loaded correctly.  */
nand_gate(x,y,t1) .
nand_gate(y,t1,t3) .
nand_gate(x,t1,t2) .
nand_gate(t2,t3,t4) .
nand_gate(t4,z,t1) .
nand_gate(t4,t1,t2) .
nand_gate(z,t1,t3) .
nand_gate(t2,t3,s) .
nand_gate(x,y,t6) .
nand_gate(z,t4,t5) .
nand_gate(t6,t5,c) .

/* Some before-extraction tests, run on the database*/
/* to ensure there are no unwanted gates present.   */

yes
?- listing(xor_gate). /* no XOR-gates listed */

yes
?- listing(full_adder). /* no full-adders listed */

yes
?- extract_full_adders.   /* invoking the algorithm */

yes
?- listing(nand_gate).   /* all NAND-gates are gone */

yes
?- listing(xor_gate).  /* all XOR-gates are gone */

yes
```

```
?- listing(full_adder).  /* one full adder was found */

full_adder(x,y,z,s,c) .

yes
?- halt.   /* finished */


J:\THESIS\CODE>exit
Script completed Mon Sep 02 12:46:22 1991
```

Appendix B. *Digital Circuit Simulation Code and Results*

B.1 Four-Bit Binary Adder Code and Simulation Results

B.1.1 Prolog Code for a Four-Bit Binary Adder

```
/*************************************************************/
/*                        addrcode.pro                       */
/*  This program exploits hierarchical structures in Prolog */
/*  by exploring digital circuit relationships. A NAND-gate */
/*  is defined as the only primitive element. Groups of     */
/*  NAND-gates are hierarchically formed into XOR-gates.    */
/*  The XOR-gates coupled with more NAND-gates will form    */
/*  full-adders.  Four full-adders will be linked to form   */
/*  one four-bit adder.                                     */

/*  The following procedure defines the primitive element,  */
/*  the NAND-gate.  Only two-input NAND-gates will be used. */
/*  Input signals will be denoted by the variables X and Y. */
/*  The output will be denoted by the variable Z. The clause*/
/*  is written in the form nand(X,Y,Z).                     */
/*************************************************************/

/****************code starts here*****************************/

   nand(0,0,1).
   nand(0,1,1).
   nand(1,0,1).
   nand(1,1,0).

/*  The following rule hierarchically defines the XOR-gate  */
/*  function in terms of NAND-gates.  Again only two-input  */
/*  XOR-gates will be considered.  Input signals            */
/*  will be denoted by X and Y, output with Z.              */

   xor(X,Y,Z) :-
```

```
    nand(X,Y,T1),
    nand(X,T1,T2),
    nand(Y,T1,T3),
    nand(T2,T3,Z).

/*  Groupings of NAND and XOR-gates are used in the          */
/*  following rule to define a full-adder. The device will   */
/*  be modeled with three input-terminals, X, Y, and Z.      */
/*  Outputs will be a sum denoted with an S and a carry      */
/*  denoted with a C.                                        */

    full_adder(X,Y,Z,C,S) :-
      nand(X,Y,T2),
      xor(X,Y,T1),
      nand(Z,T1,T3),
      xor(Z,T1,S),
      nand(T2,T3,C).

/*  The following rule defines a four-bit                    */
/*  binary-adder in terms of four full-adders.               */

     bin_adder(bin(X3,X2,X1,X0),bin(Y3,Y2,Y1,Y0),bin(C3,S3,S2,S1,S0)) :-
       full_adder(X0,Y0,0,C0,S0),
       full_adder(X1,Y1,C0,C1,S1),
       full_adder(X2,Y2,C1,C2,S2),
       full_adder(X3,Y3,C2,C3,S3).
/**************************The End**************************/
```

B.1.2 Four-Bit Binary-Adder Simulation Results

```
/*************************************************************/
/**                      addrrun.pro                        **/
/**Script session of addrcode.pro. This session contains    **/
/**several sample simul_tions of the operation of a         **/
/**four-bit binary-adder.                                   **/
/*************************************************************/

Script V1.0 session started Mon Sep 09 09:24:35 1991

Microsoft(R) MS-DOS(R) Version 4.01
            (C)Copyright Microsoft Corp 1981-1988

J:\>prolog
+-------------------------------------------+
| MS-DOS Prolog-1              Version 2.2  |
| Copyright 1983      Serial number: 0001213 |
| Expert Systems Ltd.                       |
| Oxford U.K.                               |
+-------------------------------------------+

?- [addrcode]. /** load addrcode **/addrcode consulted.

/**Test with all inputs set equal to zero**/

?- bin_adder(bin(0,0,0,0),bin(0,0,0,0),bin(C3,S3,S2,S1,S0)).

C3 = 0
S3 = 0
S2 = 0
S1 = 0
S0 = 0
More (y/n)? y
```

```
no

/**Test binary 1 added to binary 15 **/

?- bin_adder(bin(1,1,1,1),bin(0,0,0,1),bin(C3,S3,S2,S1,S0)).

C3 = 1 /** Carry bit is high, all others are low.**/
S3 = 0
S2 = 0
S1 = 0
S0 = 0
More (y/n)? y

no

/**Reverse simulation test. Given that all X inputs are set**/
/**to zero and the output is given as binary 1, what must  **/
/**value at Y have been ?**/

?- bin_adder(bin(0,0,0,0),Y,bin(0,0,0,0,1)).

Y = bin(0,0,0,1) /**Correct value**/
More (y/n)? y

no

/**Another reverse simulation. If the output was zero, what**/
/**possible input combinations are there ?**/

?- bin_adder(X,Y,bin(0,0,0,0,0)).
X = bin(0,0,0,0) /** This is the only possible input combination**/
Y = bin(0,0,0,0)
More (y/n)? y

no
```

```
/**With the output set at 1, Find all possible input **/
/**combinations.**/

?- bin_adder(X,Y,bin(0,0,0,0,1)).
X = bin(0,0,0,0) /**One of two**/
Y = bin(0,0,0,1)
More (y/n)? y
X = bin(0,0,0,1) /**two of two**/
Y = bin(0,0,0,0)
More (y/n)? y

no

/**Find all possible input combinations for an output**/
/**equal to binary 2.**/

?- bin_adder(X,Y,bin(0,0,0,1,0)).
X = bin(0,0,0,0) /**There are three possibilities**/
Y = bin(0,0,1,0)
More (y/n)? y
X = bin(0,0,1,0)
Y = bin(0,0,0,0)
More (y/n)? y
X = bin(0,0,0,1)
Y = bin(0,0,0,1)
More (y/n)? y

no
?- halt.

J:\>exit
Script completed Mon Sep 09 09:31:51 1991
```

B.2 JK Flip-Flop Code and Simulation Results

B.2.1 Prolog Code for a JK Flip-Flop

```
/*************************************************************/
/**                      jkcode.pro                         **/
/** This code simulates the operation of a jk flip-flop.    **/
/** Input sequences are assumed to be the same length.      **/
/** A list of all state transitions is returned.            **/
/** The predicate is formed as:                             **/
/**                                                         **/
/**    jksim([J|Jr],[K|Kr],Ps,[Ns|Nr]).                    **/
/**                                                         **/
/** Where: J|Jr is the list of inputs to the J-port.        **/
/**        K|Kr is the list of inputs to the K-port.        **/
/**        Ps is the present state - Note that the          **/
/**        present state must be specified inorder to       **/
/**        initiate the session.                            **/
/**        Ns|Nr is the list of state transitions which     **/
/**        correspond to the sequence of input values       **/
/**                                                         **/
/** A typical session would be:                             **/
/**                                                         **/
/** ?-jksim([1,0,1,1,1],[0,1,1,1,0,],0,Q).                 **/
/**   Q = [1,0,1,0,1]                                       **/
/**   more (y/n) y                                          **/
/**   no                                                    **/
/**                                                         **/
/*************************************************************/
/***************The code starts here************************/

    jksim([],[],_,[]).  /*boundry condition */

    jksim([J|Jr],[K|Kr],Ps,[Ns|Nr]) :-
       jkff(J,K,Ps,Ns),
       jksim(Jr,Kr,Ns,Nr).
```

```
/** The operation of the JK flip-flop is defined in boolean**/
/** terms with the following facts.                        **/

    jkff(0,0,0,0).
    jkff(0,0,1,1).
    jkff(0,1,0,0).
    jkff(0,1,1,0).
    jkff(1,0,0,1).
    jkff(1,0,1,1).
    jkff(1,1,0,1).
    jkff(1,1,1,0).

again :-
  reconsult(jkcode).
```

B.2.2 JK Flip-Flop Simulation Results

```
/*************************************************************/
/**                        jkrun.pro                        */
/** This is the script session of jkcode.pro. This session  */
/** contains several sample simulations of the operation of */
/** a JK flip-flop.                                         */
/*************************************************************/

Script V1.0 session started Tue Sep 10 11:40:29 1991

Microsoft(R) MS-DOS(R) Version 4.01
            (C)Copyright Microsoft Corp 1981-1988

A:\THESIS\CODE>prolog
+-------------------------------------------+
| MS-DOS Prolog-1             Version 2.2   |
| Copyright 1983      Serial number: 0001213  |
| Expert Systems Ltd.                       |
| Oxford U.K.                               |
+-------------------------------------------+

?- [jkcode]. /** load jkcode.pro **/
jkcode consulted.
?- jksim([0],[0],0,Q). /* beginning an exhaustive test of all
                                possible test cases.          */
Q = [0]
More (y/n)? y

no

?- jksim([0],[0],1,Q).
Q = [1]
More (y/n)? y
```

```
no
?- jksim([0],[1],0,Q).
Q = [0]
More (y/n)? y

no
?- jksim([0],[1],1,Q).
Q = [0]
More (y/n)? y

no
?- jksim([1],[0],0,Q).
Q = [1]
More (y/n)? y

no
?- jksim([1],[0],1,Q).
Q = [1]
More (y/n)? y

no
?- jksim([1],[1],0,Q).
Q = [1]
More (y/n)? y

no
?- jksim([1],[1],1,Q).
Q = [0]                              /* all cases test fine. */
More (y/n)? y

no
?- jksim([1,0,1,1,1],[0,1,1,1,0],0,Q). /*testing an input */
                                       /* sequence         */
Q = [1,0,1,0,1]
more (y/n)? y
```

```
no
?- halt.

A:\THESIS\CODE>exit
Script completed Tue Sep 10 11:45:50 1991
```

B.3 TTLS Pattern-Detection Example Derivation

B.3.1 Derivation of the Pattern-Detection Sequential Network. The derivation of the Mealy network given in Chapter 5 is discussed here for convenience. The operation of the pattern-detector is completely described by Figure B.1.

The State-table can be derived directly form the Mealy graph by filling the table in for the appropriate transitions [43]. The corresponding State-table is

Table B.1. State-table for Figure B.1.

State	State Assignment AB	Next State X = 0	Next State X = 1	Output X = 0	Output X = 1
S0	00	01	00	0	0
S1	01	11	00	0	0
S2	11	11	10	0	1
S3	10	01	11	0	0

The next state and output maps can be constructed directly from table B.1. The general Karnaugh maps of Figure B.2 must now be modified in accordance with the characteristic equation of the type of flip-flop used. The $J\overline{K}$ flip-flop used in the example. $J\overline{K}$ flip-flops are positive edge triggered flip-flops and are listed in the TTL data-book as type SN74109. See Table B.2 for the complete definition of allowable $J\overline{K}$ state transitions. The $J\overline{K}$ flip-flop specific Karnaugh maps and the output map along with the collected terms and appropriate equations are given in Figure B.3.

Table B.2. State transition table for a $J\overline{K}$ flip-flop.

J	$\overline{K}$	Ps	Ns
0	0	0	1
0	0	0	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	1	0
1	1	1	0

The logic equations derived from the maps of Figure B.3 can now be used to derive the schematic diagram of the network and the Prolog code which simulates the network's operation. Figure 5.10 gives the gate-level representation of the circuit.

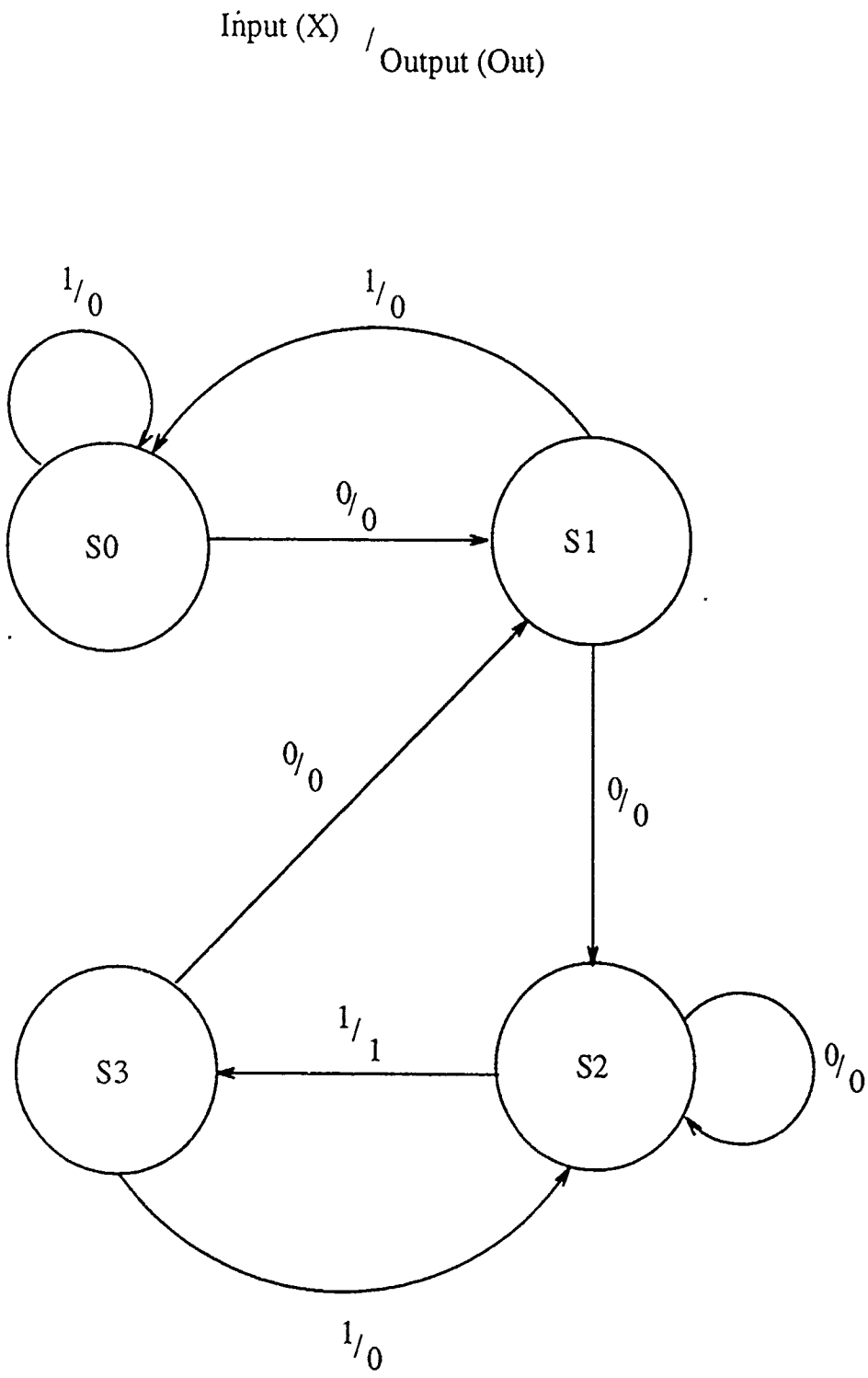


Figure B.1. Mealy state-graph for pattern-detector example

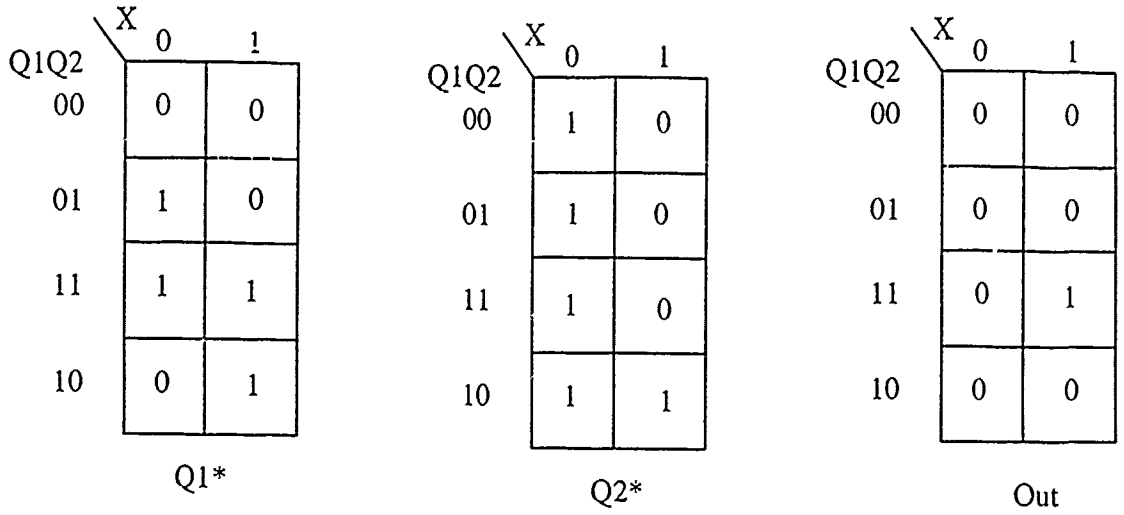
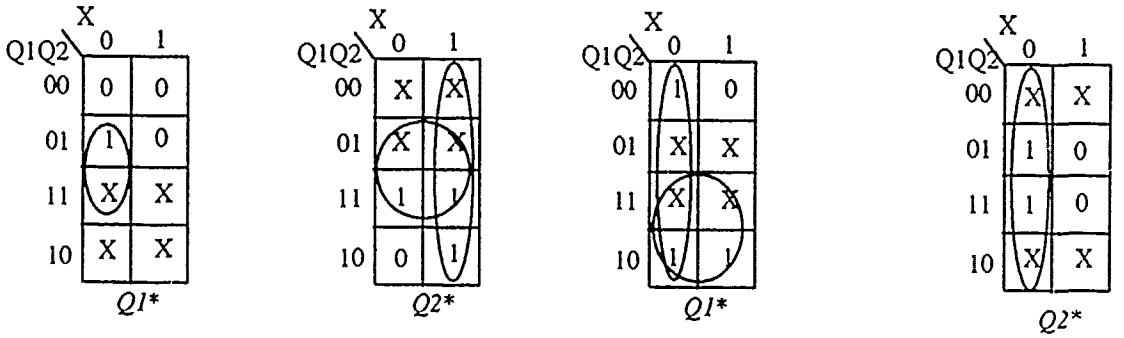


Figure B.2. General Karnaugh maps for Table B.1.



* = Next State

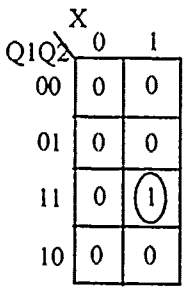


Figure B.3. $J\overline{K}$ flip-flop Karnaugh maps

B.4 TTLS Code

B.4.1 TTLS Code and Demonstration Circuit-Files

```
/*************************ttl.pro************************/
/** This file sets up the simulation.                  **/


?- write('Loading TTLSIM.PRO.'),nl,
   reconsult(ttlsim),
   write('Loading TTLDATA.PRO.'),nl,
   reconsult(ttldata),
   write('Loading SETUP.PRO'),nl,
   reconsult(setup),
   nl,
   write('Type ''go'' at the next prompt to start the process.'),nl,
   write('Be sure to end all commands with a period.'),nl,nl.
/*******************************The End*********************/
```

```
/**********************setup.pro*****************************/
/** This file sets up the simulation by supplying the user  **/
/** with some basic information concerning input files and  **/
/** asking the user which file he/she desires to load.      **/

                .
                .

go :-
   write('Two demonstration files are provided with'),nl,
   write('this simulator. The first'),
   write(','),write(' CKT1.PRO'),write(','),
   write(' describes a full-adder.'),nl,
   write('The second'),
   write(','),write(' CKT2.PRO'),write(','),
   write('describes a pattern-detector which'),nl,
   write('generates an output (1) each time the input'),
   write(' stream has at least'),nl,
   write('two zeros followed by an odd number of ones.'),nl,
   write('If you want to simulate your own circuit file;'),nl,
   write('type the file name at the prompt. Your file'),nl,
   write('must have the form ''filename.pro'' but type only'),nl,
   write('the file name at the prompt. You may return to dos'),nl,
   write('at any time by typing ''halt.'' at the prolog'),
   write(' prompt.'),nl,
   nl,
   write('Which file would you like to simulate?'),
   write(' > '), write('(ckt1./ckt2./YourFile)'),nl,
   read(File),
   procede(File).

procede(ckt1) :-
  write('Loading CKT1.PRO'),nl,
  reconsult(ckt1),
  run. /** run starts ttlsim.pro**/
```

```
procede(ckt2) :-
  write('Loading CKT2.PRO'),nl,
  reconsult(ckt2),
  run.

procede(File) :-
  write('Loading '),write(File),write('.PRO'),nl,
  reconsult(File),
  run.

/************************The End*******************************/
```

```
/************************ttlsim.pro**************************/
/** The user must number his package types with unique ic   **/
/** numbers if the total number of individual gates exceeds **/
/** the number of gates available on the ic package. For    **/
/** example if six NOR-gates were going to be used the first**/
/** four gates would come from ic1 (since an SN7402 has 4   **/
/** gates per package) and the remaining two gates from     **/
/** gates from ic2.                                         **/


/*****************************************************************/
/** run starts the simulation process by invoking the       **/
/** procedure build_circuit(QArgs). QArgs are the arguments**/
/** necessary for the simulation of memory devices. QArgs  **/
/** has the form [[[Q,NotQ],[Q+,NotQ+]],...] with two lists**/
/** of two entries for each memory device. The first list  **/
/** represents the present state and the present state's   **/
/** negation. The second list represents the future state  **/
/** and the future state's negation.                       **/
/**  set_up_state(QArgs,States) takes the list of memory   **/
/** device arguments (QArgs) and inserts the boolean values**/
/** [0,1] into the list which represents each memory       **/
/** device's future state arguments. These will later be   **/
/** shifted into the list representing the present state   **/
/** arguments immediately prior to the first cycle of the  **/
/** simulation. This will effectively clear all memory     **/
/** devices. In order for any simulation to provide        **/
/** consistent results, the memory devices must be         **/
/** initialized to a known state.                          **/
/**  The final step in the process is performed by         **/
/** evaluate_circuit(Lists,States,OutputList). Lists is the**/
/** list of input test values provided by the user. States **/
/** is discussed above. OutputList is the list of output   **/
/** values which result from the simulation.               **/
/**  The last few lines of code allow additional input     **/
```

```
/** values to be entered directly from the keyboard.       **/



run :-
  build_circuit(QArgs),
  input_sequence(Lists),
  set_up_state(QArgs,States),
  write('CLOCK CYCLE'),tab(5), /**formatting the output**/
  write('INPUT VALUES'),tab(10),
  write('OUTPUT VALUES'),nl,
  evaluate_circuit(Lists,States,0,OutputList),
  nl,
  write('Do you want to input another simulation'),
  write(' sequence? (yes./no.)'),write('>'),nl,
  read(Reply),
  ((Reply = yes,
  run_again(QArgs))
  ;
  write('simulation over')),nl,  halt. /** return to dos **/

/*****************************************************/
/**  run_again(QArgs) allows the user to run additional    **/
/** tests with new input sequences. The new sequences are  **/
/** input directly from the keyboard. Since the circuit    **/
/** whose operation is to be simulated has already been    **/
/** derived, there is very little computational overhead   **/
/** involved in these subsequent simulations; therefore the**/
/** output results are reported almost immediately to the  **/
/** user. The structured feature of the circuit's predicate**/
/** is also used here. If the user fails to input a list   **/
/** with the proper format, the procedure will fail and    **/
/** prompt for another input. The failure results from the **/
/** interpreter's inability to pattern-match               **/
/** the newly formed predicate (with the incorrectly       **/
```

```
/** formatted input list) with the old previously defined  **/
/** format.                                                **/
/**  If additional simulations are desired and the new     **/
/** sequence list is formatted properly, subsequent        **/
/** simulations are performed by clearing any memory       **/
/** devices in the circuit and reconfiguring another       **/
/** predicate (Head) with the new input values.            **/

run_again(QArgs) :-
  nl,
  write('Please enter the new sequence as a list of lists.'),nl,
  write('One sublist should be entered for each simulation '),nl,
  write('clock cycle desired. Each sublist should contain '),nl,
  write('the correct number of input values. Only the'),nl,
  write('boolean values 0 or 1 may be entered.'),nl,
  write('Enter your input now. Do not forget to'),nl,
  write('to follow your entry with a period.'),nl,
  read(NewInput),
  set_up_state(QArgs,States),
  write('CLOCK CYCLE'),tab(5), /**formatting the output**/
  write('INPUT VALUES'),tab(10),
  write('OUTPUT VALUES'),nl,
  evaluate_circuit(NewInput,States,0,OutputList),
  nl,
  write('Do you want to input another simulation'),nl,
  write(' sequence? (yes./ no.) '), write('>'),
  read(Reply),
  ((Reply = yes,
  run_again(QArgs))
  ;
  (Reply = no,
  write('simulation over'),nl,
  halt)) /** return to dos **/
  ;
  (nl,
```

```
  write('ERROR: You did not enter your new input sequence'),nl,
  write('list in the proper format. Enter the correct form '),nl,
  write('of the input sequence.'),nl,
  write('Do you want to try again? (yes./no.)'),nl,  read(NewReply),
  (NewReply = yes,
  run_again(QArgs))
  ;
  (NewReply = no,
  write('simulation over'),nl,
  halt)). /** return to dos **/



/*************************************************************/
/** set_up_state(QArgs,States) inserts the boolean values  **/
/** [0,1] into each memory device's next-state argument    **/
/** list. At a later stage, the values are transferred from **/
/** the next-state list into the present-state list,       **/
/** effectively clearing or initializing the memory devices**/
/** prior to the initiation of the simulation.             **/

set_up_state([],[]).
set_up_state([[X,Y]|Xs],[[X,[0,1]]|Zs]) :-
  set_up_state(Xs,Zs).

/*************************************************************/
/** build_circuit(QArgs) constructs a Prolog executable    **/
/** logic representation of the circuit under test from the**/
/** user provided wire-list (wire_list(L)). The wire-list  **/
/** is first converted into normal Prolog list form.       **/
/**  convert_wire_list(wire_list,WireList) performs the    **/
/** conversion by taking wire_list and removing some of the**/
/** internal Prolog "glue". This enables the modified list **/
/** WireList to be treated as a normal Prolog list. Next   **/
/** get_gates(WireList,WireList,Acc,GateList) assembles a  **/
```

```
/** list of all gates occurring in WireList into the list  **/
/** GateList. The accumulator (Acc) is used to store the   **/
/** Prolog-assigned variable wire names. If a wire name is **/
/** encountered more than once during the construction of  **/
/** the gate list, get_gates will ignore the repeated      **/
/** entry. This avoids the repeated specification of       **/
/** similar gates.                                         **/
/**  get_inargs(WireList,Acc,InArgs) examines WireList for **/
/** for any wire names which are inputs. The accumulator is**/
/** used for comparison. If a previously picked-up name    **/
/** occurs again, the name is not added to the list InArgs**/
/** in order to prevent repeat entries. The list InArgs is **/
/** built using a technique known as bottom-up. This allows**/
/** the order of the arguments in InArgs to reflect the    **/
/** order of the arguments appearance in WireList.         **/
/**  get_outargs(WireList,Acc,OutArgs) works just like     **/
/** get_inargs except that WireList is now scanned for any **/
/** output variable wire names.                            **/
/**  get_q_args(GateList,QArgs) goes through the list of   **/
/** gates (GateList) and extracts copies of the arguments  **/
/** of any memory device. These are used later to construct**/
/** the head of the circuit and to initialize the memory   **/
/** devices prior to simulation.                           **/
/**  the head (Head) of the circuit is constructed with    **/
/** "univ". Each argument in Head represents a structure   **/
/** which will be used later in the simulation. By building**/
/** Head as a composite of internal structures, this       **/
/** procedure enables the user to remove old values and    **/
/** insert new ones with simple, quick procedures.         **/
/**  unite_circuit(Head,GateList,Circuit) reapplies the    **/
/** Prolog "glue" which was removed earlier and unites     **/
/** Head with the "glued" GateList to form a circuit       **/
/** description of the original wire_list which is         **/
/** expressed in executable Prolog code. The new circuit   **/
/** (Circuit) is asserted as a rule for future use. Head   **/
```

```
/** is also asserted. It will be recalled later not for    **/
/** the particular variables which it holds but instead    **/
/** because it specifies the structure which any future    **/
/** queries to Circuit must follow.                        **/

build_circuit(QArgs) :-
  convert_wire_list(wire_list,WireList),
  write('The wire list is: '),nl,
  list_print(WireList),nl,
  get_gates(WireList,WireList,[],GateList),
  get_inargs(WireList,[],InArgs),
  get_outargs(WireList,[],OutArgs),
  get_q_args(GateList,QArgs),
  Head =..['circuit'|[InArgs,OutArgs,QArgs]],
  assert(descriptor(Head)),
  unite_circuit(Head,GateList,Circuit),
  write('The gate level circuit is: '),nl,
  assert(Circuit),
  list_print(GateList),nl.


/*************************************************************/
/** convert_wire_list(Head,GoalList) finds the body which   **/
/** belongs to Head and removes the Prolog rule "glue";     **/
/** hence allowing each goal of Body to be treated as       **/
/** though it were an element of a list.                    **/


convert_wire_list(Head,GoalList) :-
  clause(Head,Body),
  goals_to_list(Body,GoalList).


/*************************************************************/
/** goals_to_list(Structure,List) removes the internal      **/
```

```
/** Prolog rule "glue" and allows each goal of Structure   **/
/** to be treated as though it were an element of a list.  **/

goals_to_list(true,[]) :- !.
goals_to_list(`,'(First,Rest),[First|ConvertRest]) :-
  !,
  goals_to_list(Rest,ConvertRest).
goals_to_list(Goal,[Goal]).

/*************************************************************/
/** get_gates(WireList,WireList,Acc,GateList) goes through **/
/** each entry of WireList looking for an output pin.      **/
/** Once an output pin is located, get_gates then searches **/
/** for the corresponding input pins. In the case of       **/
/** devices with more than one output, all constituent     **/
/** input and output pins are located. Once a pin is       **/
/** located the corresponding wire name (expressed as a    **/
/** variable) is determined. The logic function of the     **/
/** particular device owning the pins is determined and a  **/
/** corresponding gate is formed. The gate is described as **/
/** a Prolog term with the logic  function as the functor, **/
/** the input pin wire names as input arguments, and the   **/
/** the output pin wire name or wire names (multiple       **/
/** outputs)as output arguments. For example a integrated  **/
/** circuit whose type was SN7400 with Wire1 on pin 1,     **/
/** Wire2 on Pin2, and Wire3 on Pin3 would be expressed as **/
/** nand(_51,_52,_53) where _51 represents Wire1, _52      **/
/** Wire2, and _53 Wire3.                                  **/


get_gates([],_,_,[]).

/** This rule is for five-element WireList entries         **/
/** describing a combinational device.                     **/
```

```
get_gates([[Name,SIC,OutPin,_,_]|Xs],WireList,Acc,[Gate|Rest]) :-
  not var_member(Name,Acc),
  ic(SIC,Type),
  not memory(Type),
  pins(Type,OutPin,InPinList),
  get_in_wires(InPinList,WireList,SIC,InVars),
  insert_out_pin(InVars,Name,ArgList),
  function(Type,Fcn),
  Gate =..[Fcn|ArgList],
  get_gates(Xs,WireList,[Name|Acc],Rest).


/** This rule is for three-element WireList entries        **/
/** describing a combinational device.                     **/

get_gates([[Name,SIC,OutPin]|Xs],WireList,Acc,[Gate|Rest]) :-
  not var_member(Name,Acc),
  ic(SIC,Type),
  not memory(Type),
  pins(Type,OutPin,InPinList),
  get_in_wires(InPinList,WireList,SIC,InVars),
  insert_out_pin(InVars,Name,ArgList),
  function(Type,Fcn),
  Gate =..[Fcn|ArgList],
  get_gates(Xs,WireList,[Name|Acc],Rest).

/** This rule is for five-element WireList entries         **/
/** describing a memory device.                            **/

get_gates([[Name,SIC,OutPin,_,_]|Xs],WireList,Acc,[Gate|Rest]) :-
  not var_member(Name,Acc),
  ic(SIC,Type),
  memory(Type),
  pins(Type,OutPinList,InPinList),
  var_ nber(OutPin,OutPinList),
```

```
  get_in_wires(InPinList,WireList,SIC,InVars),
  get_out_wires(OutPinList,WireList,SIC,OutVars),
  function(Type,Fcn),
  Gate =..[Fcn|[InVars,OutVars,[Qplus,NQplus]]],
  append(OutVars,Acc,NewAcc),
  get_gates(Xs,WireList,NewAcc,Rest).


/** This rule is for three-element WireList entries        **/
/** describing a memory device.                            **/


get_gates([[Name,SIC,OutPin]|Xs],WireList,Acc,[Gate|Rest]) :-
  not var_member(Name,Acc),
  ic(SIC,Type),
  memory(Type),
  pins(Type,OutPinList,InPinList),
  var_member(OutPin,OutPinList),
  get_in_wires(InPinList,WireList,SIC,InVars),
  get_out_wires(OutPinList,WireList,SIC,OutVars),
  function(Type,Fcn),
  Gate =..[Fcn|[InVars,OutVars,[Qplus,NQplus]]],
  append(OutVars,Acc,NewAcc),
  get_gates(Xs,WireList,NewAcc,Rest).


/** This rule discards WireList repeat entries.            **/

get_gates([X|Xs],WireList,Acc,Ans) :-
 get_gates(Xs,WireList,Acc,Ans).

/************************************************************/
/** get_in_wires(InPinList,WireList,SIC,Names) looks through**/
/** WireList for each occurrence of the source integrated  **/
/** circuit (SIC) and one of the members of the InPinList. **/
```

```
/** Each time a match is found, the variable name of the   **/
/** connecting wire is added to the list of names.         **/

get_in_wires([],_,_,[]).
get_in_wires([X|Xs],WireList,SIC,[Name|Names]) :-
  (member([Name,_,_,SIC,X],WireList)
  ;
  member([Name,SIC,X],WireList)),
  get_in_wires(Xs,WireList,SIC,Names).

/** if the interpreter fails to find a wire name for an    **/
/** input then there are problems. No floating inputs are  **/
/** allowed.                                               **/

get_in_wires([X|Xs],_,SIC,_) :-
  write('WARNING: FLOATING INPUT '),nl,
  write(X),write(' of '),
  write(SIC),write(' is not connected.'),nl,
  write('Correct the wire list and restart the process.'),nl,
  halt.


/*********************************************************/
/** get_out_wires(OutPinList,WireList,SIC,Names) looks    **/
/** through WireList for each occurrence of the source    **/
/** integrated circuit (SIC) and a member of OutPinList.  **/
/** Each time a match is found, the variable name of the  **/
/** connecting wire is added to the list of wires (Names). **/
/** get_out_wires is used by devices with multiple output **/
/** capabilities such as flip-flops.                      **/
```

```
get_out_wires([],_,_,[]).
get_out_wires([X|Xs],WireList,SIC,[Name|Names]) :-
  (member([Name,SIC,X,_,_],WireList)
  ;
  member([Name,SIC,X],WireList)),
  get_out_wires(Xs,WireList,SIC,Names).

/** if an available output is not used, fill it with a     **/
/** place holder. This is analogous to a floating output. **/

get_out_wires([X|Xs],WireList,SIC,[Z|Names]) :-
  get_out_wires(Xs,WireList,SIC,Names).

/*************************************************************/
/** insert_out_pin(InPins,OutPin,PinList) inserts the       **/
/** output pin (OutPin) into the list of input pin          **/
/** arguments (InPins). OutPin is inserted at the end of    **/
/** the list as by convention, the input arguments to any   **/
/** device are listed first. insert_out_pin is used for     **/
/** combinational devices only.                             **/

insert_out_pin([X],OutPin,[X,OutPin]).
insert_out_pin([X|Xs],OutPin,[X|Rest]) :-
  insert_out_pin(Xs,OutPin,Rest).


/*************************************************************/
/** get_inargs(WireList,Acc,InArgs) goes through WireList   **/
/** and finds each arity three entry which has an input pin **/
/** as its third argument. This can only be an input. The   **/
/** list of input arguments returned to the invoking procedure**/
/** is in a forward order reflecting the order of appearance**/
/** of input wire variable-names encountered as the procedure**/
/** searched through WireList. Note that the procedure      **/
/** will backtrack off the procedure "pins" looking for an  **/
```

```
/** in-pin list (InPinList) of which Pin is a member. Also  **/
/** of interest is the accumulator. In this procedure, the  **/
/** contents of the accumulator are used to accumulate the  **/
/** variable names of those input-wires which have been     **/
/** previously discovered. Upon discovery of a new input-   **/
/** wire name, a comparison is made with the contents of    **/
/** the accumulator and if the input-wire was previously    **/
/** discovered, it is rejected. This prevents the same wire **/
/** name from appearing more than once in the final list of **/
/** input wire-names. When the procedure reaches the base-  **/
/** case, the empty list is returned as a "seal" on the list**/
/** of input wire-names. The list is then returned in the   **/
/** proper order and the contents of the accumulator are    **/
/** discarded. This technique is known as constructing a    **/
/** list "bottom-up" and is used in several other procedures.**/


get_inargs([],Acc,[]).
get_inargs([[Name,SIC,Pin]|Rest],Acc,[Name|Args]) :-
  ic(SIC,Type),
  pins(Type,_,InPinList),
  member(Pin,InPinList),
  not var_member(Name,Acc),
  get_inargs(Rest,[Name|Acc],Args).

get_inargs([X|Xs],Acc,Args) :-
  get_inargs(Xs,Acc,Args).


/*************************************************************/
/** get_outargs(WireList,Acc,OutArgs) goes through WireList **/
/** and finds each arity three entry which has an output pin**/
/** as the third argument. This can only be an output. The  **/
/** list of output arguments is returned with an ordering   **/
/** which represents the order of appearance of the output  **/
```

```
/** wires in WireList. Note that the list is built          **/
/** "bottom-up".                                            **/

get_outargs([],Acc,[]).
get_outargs([[Name,SIC,Pin]|Rest],Acc,[Name|Args]) :-
  ic(SIC,Type),
  pins(Type,Pin,_),
  not var_member(Name,Acc),
  get_outargs(Rest,[Name|Acc],Args).

get_outargs([X|Xs],Acc,Args) :-
 get_outargs(Xs,Acc,Args).


/*************************************************************/
/** get_q_args(GateList,QArgs) finds each occurrence of a  **/
/** memory device in GateList and extracts the devices     **/
/** arguments. These arguments are in the form of three    **/
/** lists [[J,K],[Q,NotQ],[Q+,NotQ+]]. The J,K list is     **/
/** stripped off as this does not carry any information    **/
/** which has state-to-state significance. The other two   **/
/** lists are kept. The present-state and the next-state do**/
/** have a temporal significance and must be represented in**/
/** the head of the circuit specification as well as in the**/
/** circuit body. These two lists provide the vehicle for **/
/** the propagation of state-to-state information and      **/
/** provide the capability to simulate the memory function.**/


get_q_args([],[]).
get_q_args([X|Xs],[Rmdr|Rest]) :-
  X =..[Functr|Args],
  function(Type,Functr),
  memory(Type),
  Args = [InArgs|Rmdr],
```

```
  get_q_args(Xs,Rest).


get_q_args([X|Xs],Dummy) :-
  get_q_args(Xs,Dummy).


/*************************************************************/
/** evaluate_circuit(InputList,States,Count OutPutList) first**/
/** obtains a copy of the previously asserted circuit-head **/
/** Head. Head specified the structure which would be      **/
/** needed in order to successfully query the previously   **/
/** developed and asserted definition of the test circuit. **/
/** descriptor(circuit(InList,OutList,QList)) retrieves the**/
/** correct structure with labels for future manipulation. **/
/**  update_states(QList,States,NewStates) swaps the future**/
/** state-list of States with the present state-list of    **/
/** QList. If this is the initial run through the         **/
/** simulation, this clears all memory devices. If this is **/
/** any other cycle, update_states replaces the old present**/
/** -state values with the new future-state values thereby **/
/** setting up the memory devices for the next cycle.      **/
/** Head is now respecified as having the same functor but **/
/** the input sequence as the first argument, the still    **/
/** uninstantiated output list (OutList), and the partially**/
/** instantiated list of memory arguments (NewStates).     **/
/** Head is now ready for this particular simulation cycle.**/
/**  test_circuit(Head,NextStates,Output) invokes Head as  **/
/** an executable goal and instantiates all remaining      **/
/** variables. NextStates can now be extracted to be used  **/
/** to setup the memory devices for the next cycle and the **/
/** list output represents the simulated circuit result for**/
/** the particular input list. The simulation is repeated  **/
/** until the list of input values is exhausted. Count is  **/
/** to keep track of the number of iterations and has no   **/
```

```
/** bearing on the simulation process except to inform the **/
/** user of the output values for a particular clock cycle.**/


evaluate_circuit([],States,_,[]).
evaluate_circuit([X|Xs],States,Count,[Output|Rest]) :-
  descriptor(circuit(InList,OutList,QList)),
  update_states(QList,States,NewStates),
  Head =..['circuit'|[X,OutList,NewStates]],
  test_circuit(Head,NextStates,Output),
  Count1 is Count + 1,
  tab(5),write(Count1),tab(10), /** formats output**/
  write(X),tab(19),
  write(Output),nl,
  evaluate_circuit(Xs,NextStates,Count1,Rest).

/*************************************************************/
/** update_states(QList,States,NewStates) takes the list    **/
/** QList (which has the form [X,Y] where both X and Y are  **/
/** themselves lists) and the list States (which has the    **/
/** same general form as QList) and constructs a new list   **/
/** called NewStates which contains the second list of      **/
/** States followed by the second list of QList. The        **/
/** contribution from States is critical and represents the **/
/** next-state values of the previous simulation cycle.     **/
/** These are used as the present-state values for the next **/
/** QList contributes two variables which have no           **/
/** significance other than place holders for future        **/
/** instantiations. This swapping action acts to clear all  **/
/** memory devices on initial entry in to the procedure.    **/
/** subsequent iterations up-date each memory device by     **/
/** discarding the old present-state values and replacing   **/
/** them with the next-state values.                        **/

update_states([],[],[]).
```

```
update_states([[W,X]|Xs],[[Y,Z]|Zs],[[Z,X]|Rest]) :-
  update_states(Xs,Zs,Rest).


/*************************************************************/
/** test_circuit(Head,NextStates,Outputs) executes the      **/
/** present cycle's partial instantiation of Head. Head     **/
/** is fully instantiated after execution and those         **/
/** portions (structures) of interest (NextStates and       **/
/** Outputs) are extracted.                                 **/


test_circuit(Head,NextStates,Output) :-
  goal(Head),
  Head =..[_|[_,Output,NextStates]]. /**Only keep the pieces**/
                                     /**of interest.        **/

/*************************************************************/
/** goal(Goal) executes Goal and returns the results        **/
/** implicitly to test_circuit. When Goal is invoked, some  **/
/** of its arguments are not instantiated. After            **/
/** invocation, all of its variables are instantiated and   **/
/** although they are not passed directly to test_circuit,  **/
/** they are available to test_circuit because both goal    **/
/** and test_circuit are on the same stack.                 **/

goal(Goal) :-
  Goal.

/*************************************************************/
/** unite_circuit(Head,GateList,Circuit) constructs an      **/
/** executable Prolog rule by uniting Head with GateList.   **/

unite_circuit(Head,GateList,(Head:- Lst)):-
 goals_to_list(Lst,GateList). /* running it backwards*/
```

```
/*************************************************************/
/** var_member(X,List) determines if the variable X is a    **/
/** member of the list List. This works similar to the      **/
/** conventional member procedure with the exception that   **/
/** strict equivalence is required.                         **/

var_member(X,[Y|_]) :-
  X==Y.
var_member(X,[_|Ys]) :-
 var_member(X,Ys).

/*************************************************************/
/** the conventional member procedure.                      **/

member(X,[X|Xs]).
member(X,[Y|Ys]) :-
  member(X,Ys).

/*************************************************************/
/** the conventional append procedure.                      **/

append([],X,X).
append([X|Xs],Ys,[X|Zs]) :-
 append(Xs,Ys,Zs).

/*************************************************************/
/** pretty_print(List) prints lists to the screen with a    **/
/** more readable style.                                    **/

list_print([]).
list_print([X|Xs]) :-
  tab(5),write(X),nl,
  list_print(Xs).
```

```
/*****************************The End**********************/
```

```
/************************ttldata.pro*************************/
/** The following information was taken from a 7400 series **/
/** TTL data book.                                         **/



/*********************************************************** */
/** pins(Type,OutPin,ListOfInpins) describes the pin       **/
/** configuration of each integrated circuit in the        **/
/** data-base. This information is used to determine       **/
/** circuit connectivity.                                  **/


pins('7400',pin3,[pin1,pin2]).
pins('7400',pin6,[pin4,pin5]).
pins('7400',pin8,[pin9,pin10]).
pins('7400',pin11,[pin12,pin13]).

pins('7402',pin1,[pin2,pin3]).
pins('7402',pin4,[pin5,pin6]).
pins('7402',pin10,[pin8,pin9]).
pins('7402',pin14,[pin12,pin13]).

pins('7404',pin2,[pin1]).
pins('7404',pin4,[pin3]).
pins('7404',pin6,[pin5]).
pins('7404',pin8,[pin9]).
pins('7404',pin10,[pin11]).
pins('7404',pin12,[pin13]).

pins('7408',pin3,[pin1,pin2]).
pins('7408',pin6,[pin4,pin5]).
pins('7408',pin8,[pin9,pin10]).
pins('7408',pin11,[pin12,pin13]).
```

```
pins('7410',pin12,[pin1,pin2,pin13]).
pins('7410',pin6,[pin3,pin4,pin5]).
pins('7410',pin8,[pin9,pin10,pin11]).

pins('7432',pin3,[pin1,pin2]).
pins('7432',pin6,[pin4,pin5]).
pins('7432',pin8,[pin9,pin10]).
pins('7432',pin11,[pin12,pin13]).

pins('7486',pin3,[pin1,pin2]).
pins('7486',pin6,[pin4,pin5]).
pins('7486',pin8,[pin9,pin10]).
pins('7486',pin11,[pin12,pin13]).

pins('74109',[pin6,pin7],[pin2,pin3]).
pins('74109',[pin10,pin9],[pin14,pin13]).

/*****************************************************************/
/** function(Type,Function) denotes which type of               **/
/** integrated circuit performs which type of logic             **/
/** function.                                                   **/


function('7400',nand).
function('7402',nor).
function('7404',inv).
function('7408',and).
function('7410',nand).
function('7432',or).
function('7486',xor).
function('74109',jkbar).

/*****************************************************************/
/** memory(Type) lists those devices which belong to the        **/
```

```
/** general class of devices capable of performing a       **/
/** memory function.                                       **/

memory('74109').


/*************************************************************/
/** The following information describes gate level         **/
/** operation.                                             **/


/** jkbar([JIn,Kin],[Q,QBar],[Q+,Q+Bar])                   **/
/** jkbar is always queried with [J,K] instantiated; hence **/
/** no cuts are necessary.                                 **/

.

jkbar([0,0],[_,_],[0,1]).      /** next-state low **/
jkbar([1,1],[_,_],[1,0]).      /** next-state high **/
jkbar([0,1],[Q,QBar],[Q,QBar]). /** no change **/
jkbar([1,0],[Q,QBar],[QBar,Q]). /** toggle **/


/** three-input nand-gate operation.                       **/

nand(1,1,1,0) :- !.
nand(_,_,_,1).

/** two-input nand-gate operation.                         **/

nand(1,1,0) :- !.
nand(_,_,1).

/** two-input and-gate operation.                          **/
```

```
and(1,1,1) :- !.
and(_,_,0).

/** two-input nor-gate operation.                          **/

nor(0,0,1) :- !.
nor(_,_,0).

/** two-input xor-gate operation.                          **/

xor(A,B,1) :-
  diff(A,B),
  !.
xor(_,_,0).

/** two-input or-gate operation.                           **/

or(0,0,0) :- !.
or(_,_,1).

/** inverter operation.                                    **/

inv(0,1).
inv(1,0).

/*****************************THE END*************************/
```

Appendix C. *Specialization Code and Test Results*

C.1 Specialization Code

C.1.1 Scan, Analyze, and Specialize Code

```
/*****************************************************************/
/**                        speccode.pro                         **/
/** This file contains the Prolog code necessary to carry out **/
/** the process of specialization as described in Clocksin's  **/
/** articles on the subject. Only the primary procedures are  **/
/** defined here. Ancillary helper procedures are defined in  **/
/** the files named routines.pro and listutil.pro.            **/
/*****************************************************************/

/*********************code starts here*****************************/

/** scan(Head) starts the specialization process by breaking  **/
/** down the upper-level of the circuit and determining if    **/
/** specialization can be applied. The head and goal arguments**/
/** are broken down and compared to find any out arguments    **/
/** which are not being used. All internal variable arguments **/
/** are instantiated using the routine gensym.pro during this **/
/** check. Unmatched arguments are asserted into the database **/
/** for future use. Input goal arguments are also asserted for**/
/** future use in determining outputs which are fanned to     **/
/** other modules.                                            **/

scan(Head) :-
  Head =..[Pred|HeadArgs],
  write('HeadArgs ='),write(HeadArgs),nl,
  clause(Head,Body),
  write('Body ='),write(Body),nl,
  goals_to_list(Body,Goals),
  write('Goals ='),write(Goals),nl,
  args_direction(Goals,[],[],GoalArgsIn,GoalArgsOut),
```

```
  write('GoalArgsIn =),write(GoalArgsIn),nl,
  write('GoalArgsOut =),write(GoalArgsOut),nl,
  unmatched(GoalArgsIn,GoalArgsOut,HeadArgs,[],Unmatched),
  write('Unmatched =),write(Unmatched),nl,
  assert(args(Unmatched)),
  write('asserting '),write(GoalArgsIn),nl,
  assert(inargs(GoalArgsIn)),
  analyze(Goals,[],Circuit),
  write('Circuit =),write(Circuit).


/** analyze(Goals,Acc,Circuit) is passed the goals of any higher     **/
/** level circuit for further processing. Goals are separated into   **/
/** those which do not have unmatched out arguments and those who    **/
/** do. A goal found to contain an unmatched out argument is         **/
/** immediately sent to specialize to be processed further. Once the **/
/** specialized version of the matched goal is returned, it is added **/
/** to the original list of goals in place of the original goal. The **/
/** modified list, which now contains all original goals which did not**/
/** have unused outputs and the specialized version of those who did,**/
/** is sent through analyze again in order to propagate the removal  **/
/** through the hierarchy. Analyze continues this cycle until all    **/
/** goals present in the accumulator have all outputs utilized.      **/
/** analyze([],Acc,NewHead) is evoked when the list of unprocessed   **/
/** goals is empty. The head of the old rule is replaced with a      **/
/** new head to signify that it is a specialized version. The        **/
/** arguments for the new head are properly configured and then united**/
/** with the list of goals in Acc. Finally the direction for the new **/
/** circuit is configured and asserted into the database.            **/

/** analyze([],Acc,NewHead).
/** This section configures the new head and it arguments. Once found, **/
/** the new head is attached to the goals in Acc and asserted into the **/
/** database. The direction is also asserted for future reference.     **/
/** This is a recursive procedure and is evoked each                   **/
```

```
/** time a level of the circuit's hierarchy is processed, starting with**/
/** the lowest or base level and working up. NewHead is the name of the**/
/** replacement head generated for the highest level specialized. This **/
/** is then returned to scan.                                           **/

analyze([],Acc,NewHead):-
  write('Finishing up in analyze'),nl,
  write('Acc = '),write(Acc),nl,
  args_direction(Acc,[],[],ArgsIn,ArgsOut),
  write('ArgsIn = '),write(ArgsIn),nl,
  write('ArgsOut = '),write(ArgsOut),nl,
  remove_dups(ArgsIn,[],Temp1),
  remove_dups(ArgsOut,[],Temp2),
  rmv_all_dups(Temp1,Temp2,[],[],HInArgs,HOutArgs),
  write('HInArgs = '),write(HInArgs),nl,
  write('HOutArgs = '),write(HOutArgs),nl,
  append(HInArgs,HOutArgs,NHArgs),
  write('NewArgs = '),write(NHArgs),nl,
  gensym(foo,X),
  NewHead =..[X|NHArgs],
  write('NewHead = '),write(NewHead),nl,
  scan_acc(Acc,NewAcc),
  unite(NewHead,NewAcc,Ans),
  write('asserting '),write(Ans),nl,
  assert(Ans),
  retract(inargs(Temp)),
  write('retracting '),write(Temp),nl,
  assert(direction(NewHead,HInArgs,HOutArgs)),
  write('asserting direction'),write('('),write(NewHead),
  write(','),write(HInArgs),write(','),write(HOutArgs),
  write(')'),write(')'),nl.


/** analyze(Goals,Acc,Circuit).
/** This level of analyze looks at each goal to see if it has any    **/
```

```
/** outputs which are unused. If all outputs are used, then the goal **/
/** is stored in Acc for future use. If at least one output is not   **/
/** used then that goal is passed on to the next level of analyze for**/
/** more processing.                                                 **/

analyze([X|Rest],Acc,Circuit) :-
  write('In analyze 1 '),write(X),nl,
  direction(X,InArgs,OutArgs),
  args(Unmatched),
  write('Unmatched args = '),write(Unmatched),nl,
  not matched(OutArgs,Unmatched),
  analyze(Rest,[X|Acc],Circuit).

/** analyze(Goals,Acc,Circuit).                                      **/
/** The single goal X at this level has at least one unused output.  **/
/** All original goals which occurred prior to X are stored in Acc and  **/
/** either they don't have unused outputs at this stage in the process or**/
/** they are specialized versions of previously detected goals which did.**/
/** The goal X is sent to specialize for more processing. The specialized**/
/** version is returned via the argument Ans and appended to the rest of **/
/** the yet to be explored list "Rest". This effectively replaces the    **/
/** old predicate goal with its new specialized version. This list is    **/
/** then added to the Acc and the process started all over again as      **/
/** deleted outputs in the specialized goal returned via Ans could have  **/
/** propagated back to affect other modules at the same level in the     **/
/** hierarchy. The cycle continues until all modules have been examined  **/
/** at least once and no unused outputs exist. Note that a module may    **/
/** be either primitive or high level at this point. The analyze predicate**/
/** is not interested in the hierarchical status of modules, only the     **/
/** status of the outputs.                                           **/


analyze([X|Rest],Acc,Circuit) :-
  write('In analyze 2 '),write(X),nl,
  direction(X,InArgs,OutArgs),
```

```
  args(Unmatched),
  write('Unmatched args = '),write(Unmatched),nl,
  matched(OutArgs,Unmatched),
  specialize(X,Ans),
  write('Rest = '),write(Rest),nl,
  write('Ans = '),write(Ans),nl,
  append(Rest,Ans,Temp1),
  write('Temp1 = '),write(Temp1),nl,
  append(Acc,Temp1,Temp2),
  write('Temp2 = '),write(Temp2),nl,
  analyze(Temp2,[],Circuit).

/** specialize(X,[]).                                              **/
/** This rule deals with the modules which have been identified as  **/
/** having at least one unused output and are primitive. Modules of **/
/** this type are deleted from this level of the hierarchy. The     **/
/** deleted modules input port-designators may then be added to the **/
/** list of unmatched arguments provided no other module uses that  **/
/** designator as an input. The predicate check_args handles the    **/
/** input-argument processing for the deleted modules. The predicate **/
/** remove_dups is included in the process at this stage in order to **/
/** improve efficiency by removing repeat arguments from the list of **/
/** unmatched arguments.                                           **/

specialize(X,[]) :-
  write('In specialize 1 '),write(X),nl,
  direction(X,InArgs,OutArg),
  X =..[Pred|Args],
  primitive(Pred),
  args(Unmatched),
  write('Unmatched args ='),write(Unmatched),nl,
  matched(OutArg,Unmatched),
  inargs(StoredArgs),
  check_args(InArgs,[],Ans,StoredArgs),
  inargs(Stored),
```

```
  write('checking inargs '),write(Stored),nl,
  append(Ans,Unmatched,Temp1),
  remove_dups(Temp1,[],Temp2),
  retract(args(Unmatched)),
  write('retracting '),write(Unmatched),nl,
  write('asserting all Unmatched args '),write(Temp2),nl,
  assert(args(Temp2)).


/** specialize(X,[Ans2]).                                          **/
/** This predicate handles those goals which are not primitive and  **/
/** are in need of further processing. Note that new unmatched      **/
/** arguments are added to the list of old unmatched arguments and  **/
/** asserted into the database. A common list of unmatched arguments**/
/** is kept and applied to all layers of the hierarchy. However, the**/
/** list of input arguments (inargs) is only applicable to the layer**/
/** of the hierarchy currently being worked. These arguments are kept**/
/** in the proper place on the argument stack by use of asserta.    **/
/** Once the unmatched arguments and the new input arguments have   **/
/** been added to the database, X is then entered into for processing**/
/** at the next lower hierarchical level.                          **/

specialize(X,[Ans2]) :-
  write('In specialize 2 '),write(X),nl,
  direction(X,InArgs,OutArgs),
  X =..[Pred|Args],
  not primitive(Pred),
  write('Pred = '),write(Pred),nl,
  clause(X,Body),
  goals_to_list(Body,Goals),
  write('Goals = '),write(Goals),nl,
  args_direction(Goals,[],[],GoalArgsIn,GoalArgsOut),
  unmatched(GoalArgsIn,GoalArgsOut,Args,[],NewUnmatched),
  write('Unmatched = '),write(NewUnmatched),nl, args(Unmatched),
  write('looking at '),write(Unmatched),nl,
```

```
asserta(inargs(GoalArgsIn)),
write('asserting the following Inargs '),write(GoalArgsIn),nl,
append(Unmatched,NewUnmatched,Ans),
remove_dups(Ans,[],UnmatchedArgs),
retractall(args(Unmatched)),
write('retracting '),write(Unmatched),nl,
write('asserting '),
write(UnmatchedArgs),nl,
assert(args(UnmatchedArgs)),
analyze(Goals,[],Ans2),
write('Ans2 = '),write(Ans2),nl.


?-reconsult(gensym).
?-reconsult(listutil).
?-reconsult(routines).
?-reconsult(testcrts).


again :-
  reconsult(speccode).
```

C.1.2 Specialized Ancillary Routines

```
/*************************************************************************/
/**                        routines.pro                                 **/
/** Routines.pro has all the ancillary routines developed specifically**/
/** for Clocksin's specialization algorithm to work.                   **/
/*************************************************************************/

/** check_args(InArgs,Acc,Ans,StoredArgs).
    This predicate checks the input arguments of a module which is
    slated to be deleted to see if any particular arguments are used
    elsewhere. If not, then that particular argument can be added to
    the list of unused arguments. If the argument fans to other
    modules, then it can not be removed. However if all the other
    modules which use that particular argument are subsequently removed,
    then the argument can be removed. check_args keeps track of the
    number of modules which use any one particular argument and will
    allow that input designator to be added to the list of unmatched
    arguments only when the argument is truly unmatched. **/

check_args([],Acc,Acc,StoredArgs).
check_args([X|Rest],Acc,Ans,StoredArgs):-
  delete(X,StoredArgs,Temp1),
  var_member(X,Temp1),
  retract(inargs(StoredArgs)),
  asserta(inargs(Temp1)),
  check_args(Rest,Acc,Ans,Temp1).

check_args([X|Rest],Acc,Ans,StoredArgs):-
  check_args(Rest,[X|Acc],Ans,StoredArgs).

/** delete(X,L,Ans).
    delete X from L to give an answer which is simply the list L
    with one copy of X deleted. More X's may or may not exist in
    Ans when delete is complete. **/
```

```
delete(X,[],[]).
delete(X,[X|L],L):-
  !.
delete(X,[Y|L],[Y|NewLst]):-
  delete(X,L,NewLst).



/** matched(List1,List2).
    matched checks for membership between two lists. If any element of
    list1 matches any element of list2 the match succeeds. **/

matched([],[]).
matched([X|Rest],Unmatched) :-
  var_member(X,Unmatched),
  !,
  true.
matched([X|Rest],Unmatched) :-
  matched(Rest,Unmatched).

/** remove_dups(List,Acc,Ans).
    remove_dups will remove any repeating elements in list and return
    individual members in Ans. **/

remove_dups([],Acc,Acc).
remove_dups([X|Rest],Acc,Ans) :-
  var_member(X,Rest),
  remove_dups(Rest,Acc,Ans).

remove_dups([X|Rest],Acc,Ans) :-
  not var_member(X,Rest),
  remove_dups(Rest,[X|Acc],Ans).

/** rmv_all_dups(List1,List2,Acc1,Acc2,Ans1,Ans2).
```

```
    rmv_all_dups will search list1 and list2 for common members.
    When an element is found to be a member of both lists, that
    element and all other occurrences are deleted from both lists.
    The new lists are returned in Ans1 and Ans2. **/


rmv_all_dups([],_,Acc1,Acc2,Acc1,Acc2).
rmv_all_dups([X|Rest],OutArgs,Acc1,Acc2,Ans1,Ans2):-
  var_member(X,OutArgs),
  delete(X,OutArgs,Temp1),
  rmv_all_dups(Rest,Temp1,Acc1,Temp1,Ans1,Ans2).

rmv_all_dups([X|Rest],OutArgs,Acc1,Acc2,Ans1,Ans2):-
  rmv_all_dups(Rest,OutArgs,[X|Acc1],OutArgs,Ans1,Ans2).


/** args_direction(List1,Acc1,Acc2,Ans1,Ans2).
    This predicate takes a list of goals and separates their
    arguments into two lists, one with all input arguments and the
    other with all output arguments. Input arguments are accumulated
    in Acc1, output arguments are accumulated in Acc2. **/

args_direction([],Acc1,Acc2,Acc1,Acc2).
args_direction([X|Rest],Acc1,Acc2,ArgsIn,ArgsOut):-
  direction(X,InputArgs,OutputArgs),
  append(InputArgs,Acc1,NewAcc1),
  append(OutputArgs,Acc2,NewAcc2),
  args_direction(Rest,NewAcc1,NewAcc2,ArgsIn,ArgsOut).


/** goals_to_list(list1,list2).
    goals_to_list converts a list in Prolog internal syntax to a list
    which resembles more conventional list notation.**/

goals_to_list(true,[]) :- !.
```

```
goals_to_list(',',(First,Rest),[First|ConvertRest]) :-
  !,
  goals_to_list(Rest,ConvertRest).
  goals_to_list(Goal,[Goal]).

/** unmatched(List1,List2,List3,Acc,Ans).
    Unmatched compares List2 with List1 and List3 and stores any
    arguments from list2 which do not appear in either list1 or list3
    in Acc. Any variables which are not instantiated are fixed with a
    unique instantiation provided by gensym. **/

unmatched(_,[],_,Acc,Acc).
unmatched(GAIn,[X|Rest],HeadArgs,Acc,Ans) :-
  write(' Looking at '),write(X),nl,
  var(X),
  (var_member(X,GAIn);
   var_member(X,HeadArgs)),
  !,
  gensym('T',Y),
  X=Y,
  unmatched(GAIn,Rest,HeadArgs,Acc,Ans).
unmatched(GAIn,[X|Rest],HeadArgs,Acc,Ans) :-
  nonvar(X),
  (var_member(X,GAIn);
   var_member(X,HeadArgs)),
  !,
  unmatched(GAIn,Rest,HeadArgs,Acc,Ans).
unmatched(GAIn,[X|Rest],HeadArgs,Acc,Ans) :-
  gensym('T',Y),
  X=Y,
  unmatched(GAIn,Rest,HeadArgs,[X|Acc],Ans).


/** var_member(Element,List).
    works the same as regular membership except this procedure will
```

```
    also work for variables. **/

var_member(X,[Y|_]) :-
  X==Y.
var_member(X,[_|L]) :-
  var_member(X,L).

/** scan_acc(List1,List2) will remove all terms in list1 which
    have an arity of zero and return a new list, List2 of the
    surviving terms. **/

scan_acc([],[]).
scan_acc([X|Rest1],Rest2) :-
  X =..[Pred|[]],
  scan_acc(Rest1,Rest2).
scan_acc([X|Rest1],[X|Rest2]) :-
  scan_acc(Rest1,Rest2).


/** unite(Element,List,NewRule).
    unite will connect a predicate (NewHead) with a set of goals
    (ValidGoals) to form a new rule. **/

unite(NewHead,ValidGoals,(NewHead :- ValidGoals)) :- !.
```

C.1.3 General Utility Programs

```
/*****************************************************************/
/**                        listutil.pro                         **/
/** All procedures listed here are standard utility procedures. **/
/** used by the files speccode.pro or routines.pro.             **/
/*****************************************************************/

member(X,[X|_]).
member(X,[_|Tail]):-
   member(X,Tail).

append([],L,L).
append([X|Tail],M,[X|XS]):-
   append(Tail,M,XS).
```

C.1.4 Gensym Code

```
/************************ GENSYM.pro ****************************/
/**                                                            **/
/** 'gensym(Root,Atom)' creates a new atom, which begins with  **/
/** the specified root and ends with a unique integer.         **/
/** Typical session:                                           **/
/**      ?- gensym(foo,X).                                     **/
/**      X = foo1                                              **/
/**      More (y/n)? y                                         **/
/**      no                                                    **/
/**                                                            **/
/**      ?- gensym(foo,X).                                     **/
/**      X = foo2                                              **/
/**      More (y/n)? y                                         **/
/**      no                                                    **/
/**                                                            **/
/** This code is taken from the Clocksin & Mellish book,       **/
/** Programming In Prolog, Section 7.8.                        **/
/**                                                            **/
********************************************************************/

gensym(Root,Atom) :-
  get_num(Root,Num),
  name(Root,Name1),
  integer_name(Num,Name2),
  append(Name1,Name2,Name),
  name(Atom,Name).

get_num(Root,Num) :-
  retract(current_num(Root,Num1)),
  !,
  Num is Num1 + 1,
  asserta(current_num(Root,Num)).
get_num(Root,1) :-
```

```
  asserta(current_num(Root,1)).

/* Convert from an integer to a list of characters. */

integer_name(Int,List) :-
  integer_name2(Int,[],List).

integer_name2(I,SoFar,[C|SoFar]) :-
  I < 10,
  !,
  C is I + 48.
integer_name2(I,SoFar,List) :-
  TopHalf    is I  // 10,    /* Some Prologs may call this '/' */
  BottomHalf is I mod 10,
  C is BottomHalf + 48,
  integer_name2(TopHalf,[C|SoFar],List).

again :-
  reconsult(gensym).


?- reconsult(listutil).     /* Needed for the call to 'append' */
```

C.1.5 Prolog Code Definitions of Test Circuits

```
/*************************************************************/
/**                    testcrt.pro                          **/
/** This file contains Prolog code which defines several    **/
/*  circuits used to test the program which implements      **/
/** Clocksin's specialization process.                      **/
/*************************************************************/

/**  Test1 uses a circuit which has unused primitive gates **/
/** at the highest level in the hierarchy. These unused    **/
/** primitive gates are fed by other hierarchical modules. **/
/** which may or may not need to be specialized when the   **/
/** unused primitive modules are removed.                  **/

test1 :-
   scan(circuit1(a,b,c,as,s,co)).

circuit1(A,B,C,As,S,Co) :-
 halfadd(B,C,T1,T2),
 halfadd(A,T1,S,Co),
 xor_gate(A,As,T3),
 and_gate(T1,T3,T4),
 or_gate(T2,T4,Unused).

/**  Test 2 looks at a circuit which has a hierarchical  **/
/** module consisting of other hierarchical and primitive**/
/** modules. The subordinate hierarchical module does not **/
/** have any unused outputs and can be completely eliminated. **/

test2 :-
  scan(circuit2(a,b,c,as,s,co)).

circuit2(A,B,C,As,S,Co) :-
 halfadd(B,C,Unused1,Unused2),
```

```
 halfadd(A,T3,S,Co),
 xor_gate(A,As,T3).



/** Test 3 inputs a circuit where every output is used and  **/
/** no specialization is possible.                          **/

test3 :-
  scan(circuit3(a,b,c,as,s,co)).

circuit3(A,B,C,As,S,Co) :-
 halfadd(B,C,T1,T2),
 halfadd(A,T1,S,Co),
 xor_gate(A,As,T3),
 and_gate(T1,T3,T4),
 or_gate(T2,T4,Co).

/**  Test 4 is Clocksin's test as given in the article **/

test4 :-
   scan(twobit(a1,b1,a2,b2,c,as,s1,s2)).

twobit(A1,B1,A2,B2,C,As,S1,S2) :-
 addsub(A1,B1,C,As,S1,T),
 addsub(A2,B2,T,As,S2,Unused).


addsub(A,B,C,As,S,Co) :-
 halfadd(B,C,T1,T2),
 halfadd(A,T1,S,Unused),
 xor_gate(A,As,T3),
 and_gate(T1,T3,T4),
 or_gate(T2,T4,Co).
```

```
halfadd(A,B,S,C) :-
 xor_gate(A,B,S),
 nand_gate(A,B,T),
 nnot_gate(T,C).


/** direction(Module,InputPorts,OutputPorts) is used to        **/
/** differentiate input from output ports. direction only needs **/
/** to be defined for lower-level generic modules.              **/


direction(addsub(A,B,C,As,S,Co),[A,B,C,As],[S,Co]).
direction(halfadd(A,B,S,C),[A,B],[S,C]).
direction(or_gate(A,B,Out),[A.B],[Out]).
direction(xor_gate(A,B,Out),[A,B],[Out]).
direction(and_gate(A,B,Out),[A,B],[Out]).
direction(nnot_gate(A,B),[A],[B]).
direction(nand_gate(A,B,C),[A,B],[C]).

/** primitive(Gate) identifies gates which are at the lowest  **/
/** level of the circuit hierarchy.                           **/

primitive(or_gate).
primitive(xor_gate).
primitive(and_gate).
primitive(nnot_gate).
primitive(nand_gate).
```

C.2 Specialization Test Results

C.2.1 Test 1 Code and Results

```
/*************************************************************/
/**                    test1 results                        **/
/**  Test1 uses a circuit which has unused primitive gates **/
/** at the highest level in the hierarchy. These unused    **/
/** primitive gates are fed by other hierarchical modules. **/
/** which may or may not need to be specialized when the   **/
/** unused primitive modules are removed.                  **/
/*************************************************************/

/*******************test1 circuit definition***************/
    test1 :-
      scan(circuit1(a,b,c,as,s,co)).

    circuit1(A,B,C,As,S,Co) :-
      halfadd(B,C,T1,T2),
      halfadd(A,T1,S,Co),
      xor_gate(A,As,T3),
      and_gate(T1,T3,T4),
      or_gate(T2,T4,unused).

/** The following generic submodule definitions are needed to **/
/** define the subcircuit structure and are explained in      **/
/** section C.1.5.                                            **/



    halfadd(A,B,S,C) :-
      xor_gate(A,B,S),
      nand_gate(A,B,T),
      nnot_gate(T,C).
```

```
    direction(halfadd(A,B,S,C),[A,B],[S,C]).
    direction(or_gate(A,B,Out),[A,B],[Out]).
    direction(xor_gate(A,B,Out),[A,B],[Out]).
    direction(and_gate(A,B,Out),[A,B],[Out]).
    direction(nnot_gate(A,B),[A],[B]).
    direction(nand_gate(A,B,C),[A,B],[C]).

    primitive(or_gate).
    primitive(xor_gate).
    primitive(and_gate).
    primitive(nnot_gate).
    primitive(nand_gate).

/**********************Test Results************************/


Script V1.0 session started Mon Oct 14 13:22:55 1991

Microsoft(R) MS-DOS(R) Version 4.01
             (C)Copyright Microsoft Corp 1981-1988

A:\THESIS\CODE>prolog


+-------------------------------------------+
| MS-DOS Prolog-1              Version 2.2  |
| Copyright 1983      Serial number: 0001213 |
| Expert Systems Ltd.                       |
| Oxford U.K.                               |
+-------------------------------------------+

?- [speccode].
speccode consulted.
?- test1.
HeadArgs =[a,b,c,as,s,co]
```

```
Body =halfadd(b,c,_102,_103),halfadd(a,_102,s,co),xor_gate(a,as,_118),
and_gate(_102,_118,_125),or_gate(_103,_125,_132)

Goals =[halfadd(b,c,_102,_103),halfadd(a,_102,s,co),xor_gate(a,as,_118),
and_gate(_102,_118,_125),or_gate(_103,_125,_132)]

GoalArgsIn =[_103,_125,_102,_118,a,as,a,_102,b,c]
GoalArgsOut =[_132,_125,_118,s,co,_102,_103]
 Looking at _132
 Looking at _125
 Looking at _118
 Looking at s
 Looking at co
 Looking at _102
 Looking at _103
Unmatched =[T1]
asserting [T5,T2,T4,T3,a,as,a,T4,b,c]
In analyze 1 halfadd(b,c,T4,T5)
Unmatched args = [T1]
In analyze 1 halfadd(a,T4,s,co)
Unmatched args = [T1]
In analyze 1 xor_gate(a,as,T3)
Unmatched args = [T1]
In analyze 1 and_gate(T4,T3,T2)
Unmatched args = [T1]
In analyze 1 or_gate(T5,T2,T1)
Unmatched args = [T1]
In analyze 2 or_gate(T5,T2,T1)
Unmatched args = [T1]
In specialize 1 or_gate(T5,T2,T1)
Unmatched args =[T1]
In check_args1
X = T5
Rest = [T2]
```

```
stored inargs = [T5,T2,T4,T3,a,as,a,T4,b,c]
deleting T5
from [T5,T2,T4,T3,a,as,a,T4,b,c]
Temp1 = [T2,T4,T3,a,as,a,T4,b,c]
in check_args 2
adding T5
to Acc []
In check_args1
X = T2
Rest = []
stored inargs = [T5,T2,T4,T3,a,as,a,T4,b,c]
deleting T2
from [T5,T2,T4,T3,a,as,a,T4,b,c]
Temp1 = [T5,T4,T3,a,as,a,T4,b,c]
in check_args 2
adding T2
to Acc [T5]
checking inargs [T5,T2,T4,T3,a,as,a,T4,b,c]
retracting [T1]
asserting all Unmatched args [T1,T5,T2]
Rest = []
Ans = []
Temp1 = []

Temp2 = [and_gate(T4,T3,T2),xor_gate(a,as,T3),halfadd(a,T4,s,co),
halfadd(b,c,T4,T5)]

In analyze 1 and_gate(T4,T3,T2)
Unmatched args = [T1,T5,T2]
In analyze 2 and_gate(T4,T3,T2)
Unmatched args = [T1,T5,T2]
In specialize 1 and_gate(T4,T3,T2)
Unmatched args =[T1,T5,T2]
In check_args1
X = T4
```

```
Rest = [T3]
stored inargs = [T5,T2,T4,T3,a,as,a,T4,b,c]
deleting T4
from [T5,T2,T4,T3,a,as,a,T4,b,c]
Temp1 = [T5,T2,T3,a,as,a,T4,b,c]
checking membership of T4
In [T5,T2,T3,a,as,a,T4,b,c]
retracting [T5,T2,T4,T3,a,as,a,T4,b,c]
asserting [T5,T2,T3,a,as,a,T4,b,c]
In check_args1
X = T3
Rest = []
stored inargs = [T5,T2,T3,a,as,a,T4,b,c]
deleting T3
from [T5,T2,T3,a,as,a,T4,b,c]
Temp1 = [T5,T2,a,as,a,T4,b,c]
in check_args 2
adding T3
to Acc []
checking inargs [T5,T2,T3,a,as,a,T4,b,c]
retracting [T1,T5,T2]
asserting all Unmatched args [T2,T5,T1,T3]
Rest = [xor_gate(a,as,T3),halfadd(a,T4,s,co),halfadd(b,c,T4,T5)]
Ans = []
Temp1 = [xor_gate(a,as,T3),halfadd(a,T4,s,co),halfadd(b,c,T4,T5)]
Temp2 = [xor_gate(a,as,T3),halfadd(a,T4,s,co),halfadd(b,c,T4,T5)]
In analyze 1 xor_gate(a,as,T3)
Unmatched args = [T2,T5,T1,T3]
In analyze 2 xor_gate(a,as,T3)
Unmatched args = [T2,T5,T1,T3]
In specialize 1 xor_gate(a,as,T3)
Unmatched args =[T2,T5,T1,T3]
In check_args1
X = a
Rest = [as]
```

```
stored inargs = [T5,T2,T3,a,as,a,T4,b,c]
deleting a
from [T5,T2,T3,a,as,a,T4,b,c]
Temp1 = [T5,T2,T3,as,a,T4,b,c]
checking membership of a
In [T5,T2,T3,as,a,T4,b,c]
re -acting [T5,T2,T3,a,as,a,T4,b,c]
asserting [T5,T2,T3,as,a,T4,b,c]
In check_args1
X = as
Rest = []
stored inargs = [T5,T2,T3,as,a,T4,b,c]
deleting as
from [T5,T2,T3,as,a,T4,b,c]
Temp1 = [T5,T2,T3,a,T4,b,c]
in check_args 2
adding as
to Acc []
checking inargs [T5,T2,T3,as,a,T4,b,c]
retracting [T2,T5,T1,T3]
asserting all Unmatched args [T3,T1,T5,T2,as]
Rest = [halfadd(a,T4,s,co),halfadd(b,c,T4,T5)]
Ans = []
Temp1 = [halfadd(a,T4,s,co),halfadd(b,c,T4,T5)]
Temp2 = [halfadd(a,T4,s,co),halfadd(b,c,T4,T5)]
In analyze 1 halfadd(a,T4,s,co)
Unmatched args = [T3,T1,T5,T2,as]
In analyze 1 halfadd(b,c,T4,T5)
Unmatched args = [T3,T1,T5,T2,as]
In analyze 2 halfadd(b,c,T4,T5)
Unmatched args = [T3,T1,T5,T2,as]
In specialize 1 halfadd(b,c,T4,T5)
In specialize 2 halfadd(b,c,T4,T5)
Pred = halfadd
Goals = [xor_gate(b,c,T4),nand_gate(b,c,_1061),nnot_gate(_1061,T5)]
```

```
 Looking at T5
 Looking at _1061
 Looking at T4
Unmatched = []
looking at [T3,T1,T5,T2,as]
asserting the following Inargs [T6,b,c,b,c]
retracting [T3,T1,T5,T2,as]
asserting [as,T2,T5,T1,T3]
In analyze 1 xor_gate(b,c,T4)
Unmatched args = [as,T2,T5,T1,T3]
In analyze 1 nand_gate(b,c,T6)
Unmatched args = [as,T2,T5,T1,T3]
In analyze 1 nnot_gate(T6,T5)
Unmatched args = [as,T2,T5,T1,T3]
In analyze 2 nnot_gate(T6,T5)
Unmatched args = [as,T2,T5,T1,T3]
In specialize 1 nnot_gate(T6,T5)
Unmatched args =[as,T2,T5,T1,T3]
In check_args1
X = T6
Rest = []
stored inargs = [T6,b,c,b,c]
deleting T6
from [T6,b,c,b,c]
Temp1 = [b,c,b,c]
in check_args 2
adding T6
to Acc []
checking inargs [T6,b,c,b,c]
retracting [as,T2,T5,T1,T3]
asserting all Unmatched args [T3,T1,T5,T2,as,T6]
Rest = []
Ans = []
Temp1 = []
Temp2 = [nand_gate(b,c,T6),xor_gate(b,c,T4)]
```

```
In analyze 1 nand_gate(b,c,T6)
Unmatched args = [T3,T1,T5,T2,as,T6]
In analyze 2 nand_gate(b,c,T6)
Unmatched args = [T3,T1,T5,T2,as,T6]
In specialize 1 nand_gate(b,c,T6)
Unmatched args =[T3,T1,T5,T2,as,T6]
In check_args1
X = b
Rest = [c]
stored inargs = [T6,b,c,b,c]
deleting b
from [T6,b,c,b,c]
Temp1 = [T6,c,b,c]
checking membership of b
In [T6,c,b,c]
retracting [T6,b,c,b,c]
asserting [T6,c,b,c]
In check_args1
X = c
Rest = []
stored inargs = [T6,c,b,c]
deleting c
from [T6,c,b,c]
Temp1 = [T6,b,c]
checking membership of c
In [T6,b,c]
retracting [T6,c,b,c]
asserting [T6,b,c]
checking inargs [T6,b,c]
retracting [T3,T1,T5,T2,as,T6]
asserting all Unmatched args [T6,as,T2,T5,T1,T3]
Rest = [xor_gate(b,c,T4)]
Ans = []
Temp1 = [xor_gate(b,c,T4)]
Temp2 = [xor_gate(b,c,T4)]
```

```
In analyze 1 xor_gate(b,c,T4)
Unmatched args = [T6,as,T2,T5,T1,T3]
Finishing up in analyze
Acc = [xor_gate(b,c,T4)]
ArgsIn = [b,c]
ArgsOut = [T4]
HInArgs = [b,c]
HOutArgs = [T4]
NewArgs = [b,c,T4]
NewHead = foo1(b,c,T4)
asserting foo1(b,c,T4):-[xor_gate(b,c,T4)]
retracting [T6,b,c]
asserting direction(foo1(b,c,T4),[b,c],[T4]))
Ans2 = foo1(b,c,T4)
Rest = []
Ans = [foo1(b,c,T4)]
Temp1 = [foo1(b,c,T4)]
Temp2 = [halfadd(a,T4,s,co),foo1(b,c,T4)]
In analyze 1 halfadd(a,T4,s,co)
Unmatched args = [T6,as,T2,T5,T1,T3]
In analyze 1 foo1(b,c,T4)
Unmatched args = [T6,as,T2,T5,T1,T3]
Finishing up in analyze
Acc = [foo1(b,c,T4),halfadd(a,T4,s,co)]
ArgsIn = [a,T4,b,c]
ArgsOut = [s,co,T4]
HInArgs = [a,b,c]
HOutArgs = [co,s]
NewArgs = [a,b,c,co,s]
NewHead = foo2(a,b,c,co,s)
asserting foo2(a,b,c,co,s):-[foo1(b,c,T4),halfadd(a,T4,s,co)]
retracting [T5,T2,T3,as,a,T4,b,c]
asserting direction(foo2(a,b,c,co,s),[a,b,c],[co,s]))
Circuit =foo2(a,b,c,co,s)
yes
```

```
?- listing(foo1).
foo1(b,c,'T4') :-
        [xor_gate(b,c,'T4')] .


yes
?- halt.

A:\THESIS\CODE>exit
Script completed Mon Oct 14 13:24:33 1991

/********************End of Test Results*********************/
```

The before and after circuit diagrams are shown in Figures C.1 and C.2. Note that the circuit diagram for the specialized version matches the definition of "foo2" as reported to the screen. The internally defined connections denoted by a "T" designation may change during the algorithm's execution however, the change is consistently propagated throughout the circuit in order to preserve the correctness of the interconnections. Test1 differs from Clocksin's test example, given in this appendix as test 4, in that the carry output, Co, is specified as being used at the uppermost level while the unused output, T5, remains hidden to the user.

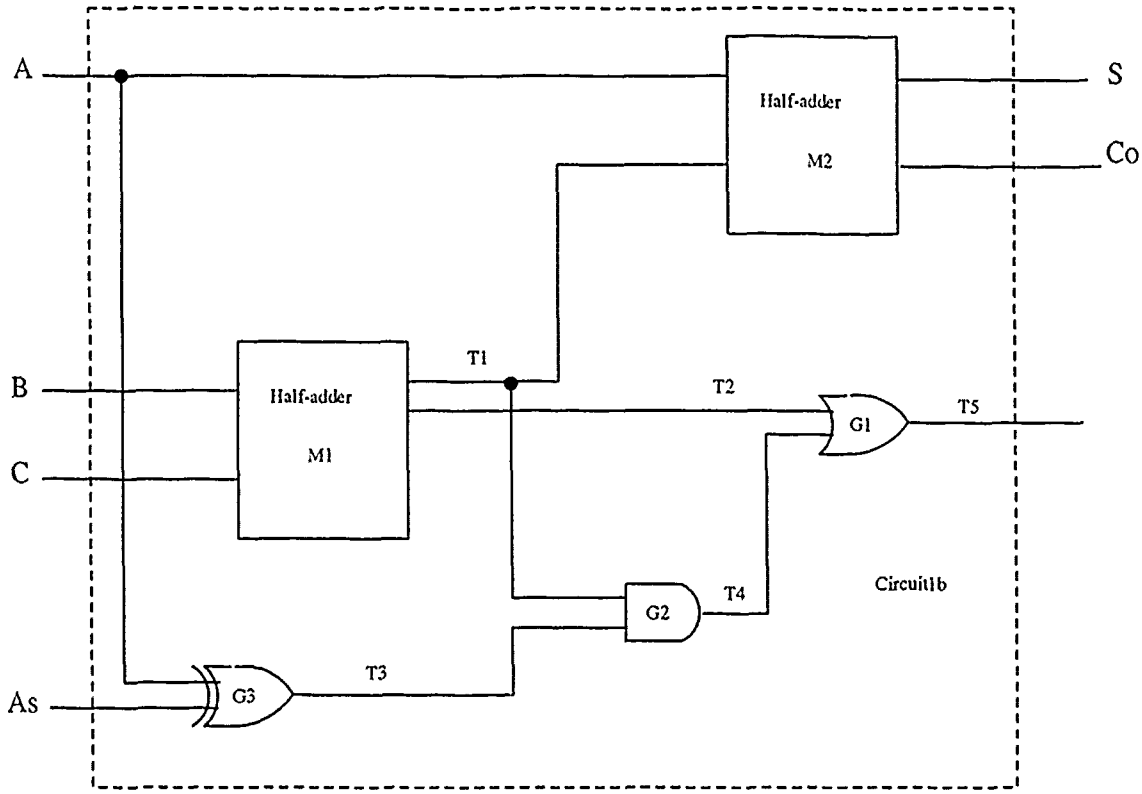


Figure C.1. Circuit schematic for test 1 before specialization.

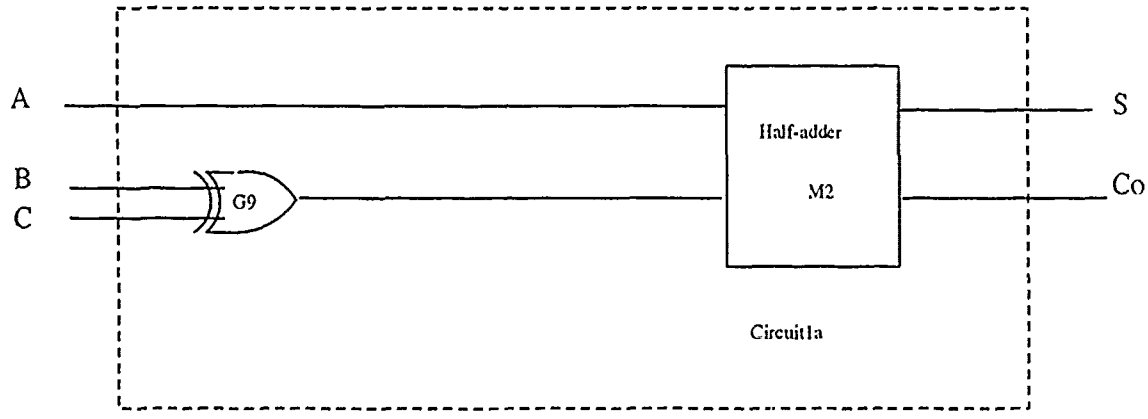


Figure C.2. Circuit schematic for test 1 after specialization.

C.2.2 Test 2 Code and Results

```
/*************************************************************/
/**                    test2 results                        **/
/**  Test 2 looks at a circuit which has a hierarchical     **/
/** module consisting of other hierarchical and primitive   **/
/** modules. The subordinate hierarchical module does not   **/
/** have any unused outputs and can be completely eliminated. **/
/*************************************************************/

/********************test2 circuit definition*******************/


     test2 :-
       scan(circuit2(a,b,c,as,s,co)).

     circuit2(A,B,C,As,S,Co) :-
       halfadd(B,C,Unused1,Unused2),
       halfadd(A,T3,S,Co),
       xor_gate(A,As,T3).

/** The following submodule definitions will be needed in   **/
/** order to execute test2. Section C.1.5 further explains  **/
/** the code.                                               **/

     halfadd(A,B,S,C) :-
       xor_gate(A,B,S),
       nand_gate(A,B,T),
       nnot_gate(T,C).


     direction(halfadd(A,B,S,C),[A,B],[S,C]).
     direction(xor_gate(A,B,Out),[A,B],[Out]).
     direction(nnot_gate(A,B),[A],[B]).
     direction(nand_gate(A,B,C),[A,B],[C]).
```

```
     primitive(xor_gate).
     primitive(nand_gate).
     primitive(nnot_gate).

/************************Test Results************************/


Script V1.0 session started Mon Oct 14 14:28:43 1991

Microsoft(R) MS-DOS(R) Version 4.01
             (C)Copyright Microsoft Corp 1981-1988

A:\THESIS\CODE>prolog


+-------------------------------------------+
| MS-DOS Prolog-1              Version 2.2  |
| Copyright 1983       Serial number: 0001213  |
| Expert Systems Ltd.                       |
| Oxford U.K.                               |
+-------------------------------------------+

?- [speccode].
speccode consulted
?- test2.
HeadArgs =[a,b,c,as,s,co]
Body =halfadd(b,c,_102,_103),halfadd(a,_109,s,co),xor_gate(a,as,_109)
Goals =[halfadd(b,c,_102,_103),halfadd(a,_109,s,co),xor_gate(a,as,_109)]
GoalArgsIn =[a,as,a,_109,b,c]
GoalArgsOut =[_109,s,co,_102,_103]
 Looking at _109
 Looking at s
 Looking at co
 Looking at _102
```

```
 Looking at _103
Unmatched =[T3,T2]
asserting [a,as,a,T1,b,c]
In analyze 1 halfadd(b,c,T2,T3)
Unmatched args = [T3,T2]
In analyze 2 halfadd(b,c,T2,T3)
Unmatched args = [T3,T2]
In specialize 1 halfadd(b,c,T2,T3)
In specialize 2 halfadd(b,c,T2,T3)
Pred = halfadd
Goals = [xor_gate(b,c,T2),nand_gate(b,c,_368),nnot_gate(_368,T3)]
 Looking at T3
 Looking at _368
 Looking at T2
Unmatched = []
looking at [T3,T2]
asserting the following Inargs [T4,b,c,b,c]
retracting [T3,T2]
asserting [T2,T3]
In analyze 1 xor_gate(b,c,T2)
Unmatched args = [T2,T3]
In analyze 2 xor_gate(b,c,T2)
Unmatched args = [T2,T3]
In specialize 1 xor_gate(b,c,T2)
Unmatched args =[T2,T3]
In check_args1
X = b
Rest = [c]
stored inargs = [T4,b,c,b,c]
deleting b
from [T4,b,c,b,c]
Temp1 = [T4,c,b,c]
checking membership of b
In [T4,c,b,c]
retracting [T4,b,c,b,c]
```

```
asserting [T4,c,b,c]
In check_args1
X = c
Rest = []
stored inargs = [T4,c,b,c]
deleting c
from [T4,c,b,c]
Temp1 = [T4,b,c]
checking membership of c
In [T4,b,c]
retracting [T4,c,b,c]
asserting [T4,b,c]
checking inargs [T4,b,c]
retracting [T2,T3]
asserting all Unmatched args [T3,T2]
Rest = [nand_gate(b,c,T4),nnot_gate(T4,T3)]
Ans = []
Temp1 = [nand_gate(b,c,T4),nnot_gate(T4,T3)]
Temp2 = [nand_gate(b,c,T4),nnot_gate(T4,T3)]
In analyze 1 nand_gate(b,c,T4)
Unmatched args = [T3,T2]
In analyze 1 nnot_gate(T4,T3)
Unmatched args = [T3,T2]
In analyze 2 nnot_gate(T4,T3)
Unmatched args = [T3,T2]
In specialize 1 nnot_gate(T4,T3)
Unmatched args =[T3,T2]
In check_args1
X = T4
Rest = []
stored inargs = [T4,b,c]
deleting T4
from [T4,b,c]
Temp1 = [b,c]
in check_args 2
```

```
adding T4
to Acc []
checking inargs [T4,b,c]
retracting [T3,T2]
asserting all Unmatched args [T2,T3,T4]
Rest = []
Ans = []
Temp1 = []
Temp2 = [nand_gate(b,c,T4)]
In analyze 1 nand_gate(b,c,T4)
Unmatched args = [T2,T3,T4]
In analyze 2 nand_gate(b,c,T4)
Unmatched args = [T2,T3,T4]
In specialize 1 nand_gate(b,c,T4)
Unmatched args =[T2,T3,T4]
In check_args1
X = b
Rest = [c]
stored inargs = [T4,b,c]
deleting b
from [T4,b,c]
Temp1 = [T4,c]
in check_args 2
adding b
to Acc []
In check_args1
X = c
Rest = []
stored inargs = [T4,b,c]
deleting c
from [T4,b,c]
Temp1 = [T4,b]
in check_args 2
adding c
to Acc [b]
```

```
checking inargs [T4,b,c]
retracting [T2,T3,T4]
asserting all Unmatched args [T4,T3,T2,b,c]
Rest = []
Ans = []
Temp1 = []
Temp2 = []
Finishing up in analyze
Acc = []
ArgsIn = []
ArgsOut = []
HInArgs = []
HOutArgs = []
NewArgs = []
NewHead = foo1
asserting foo1:-[]
retracting [T4,b,c]
asserting direction(foo1,[],[]))
Ans2 = foo1
Rest = [halfadd(a,T1,s,co),xor_gate(a,as,T1)]
Ans = [foo1]
Temp1 = [halfadd(a,T1,s,co),xor_gate(a,as,T1),foo1]
Temp2 = [halfadd(a,T1,s,co),xor_gate(a,as,T1),foo1]
In analyze 1 halfadd(a,T1,s,co)
Unmatched args = [T4,T3,T2,b,c]
In analyze 1 xor_gate(a,as,T1)
Unmatched args = [T4,T3,T2,b,c]
In analyze 1 foo1
Unmatched args = [T4,T3,T2,b,c]
Finishing up in analyze
Acc = [foo1,xor_gate(a,as,T1),halfadd(a,T1,s,co)]
ArgsIn = [a,T1,a,as]
ArgsOut = [s,co,T1]
HInArgs = [a,as]
HOutArgs = [co,s]
```

```
NewArgs = [a,as,co,s]
NewHead = foo2(a,as,co,s)
asserting foo2(a,as,co,s):-[xor_gate(a,as,T1),halfadd(a,T1,s,co)]
retracting [a,as,a,T1,b,c]
asserting direction(foo2(a,as,co,s),[a,as],[co,s]))
Circuit =foo2(a,as,co,s)
yes
?- halt.

A:\THESIS\CODE>exit
Script completed Mon Oct 14 14:29:41 1991
/************************End of Test Results******************/
```

The before and after circuit diagrams are shown in Figures C.3 and C.4 respectively. Note that the redundant higher-level module, the full-adder was successfully removed.

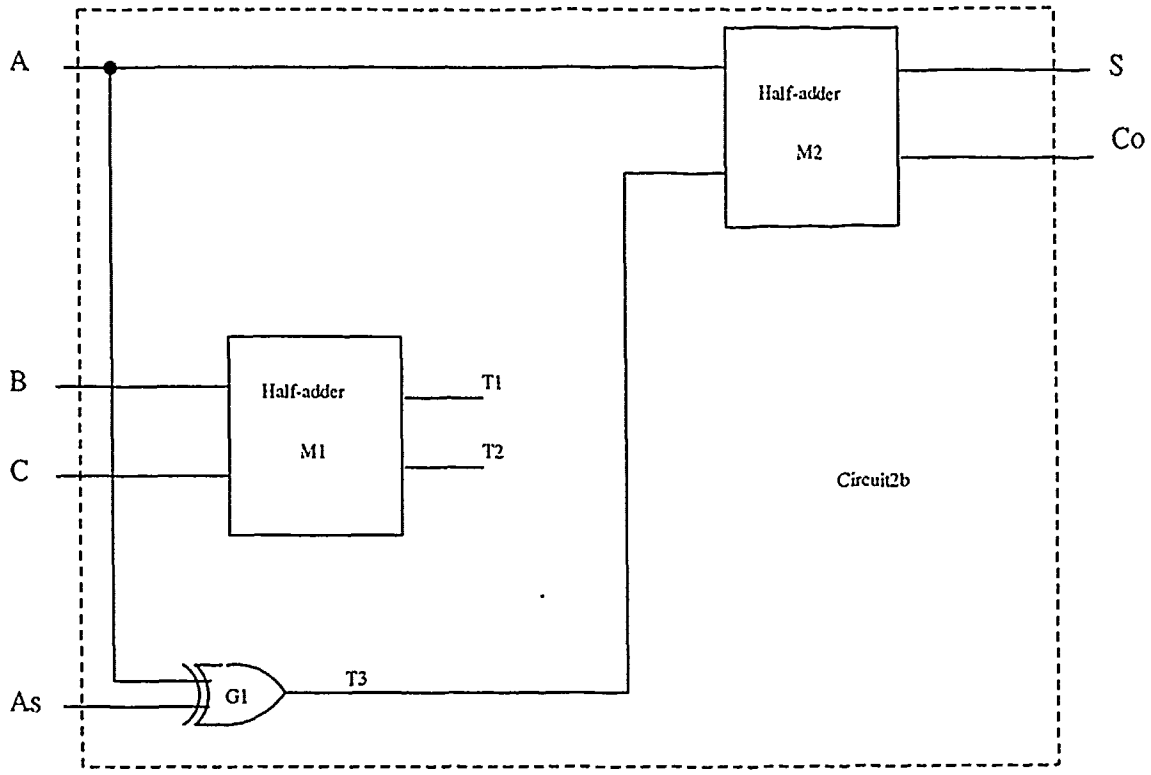


Figure C.3. Circuit schematic for test 2 before specialization.

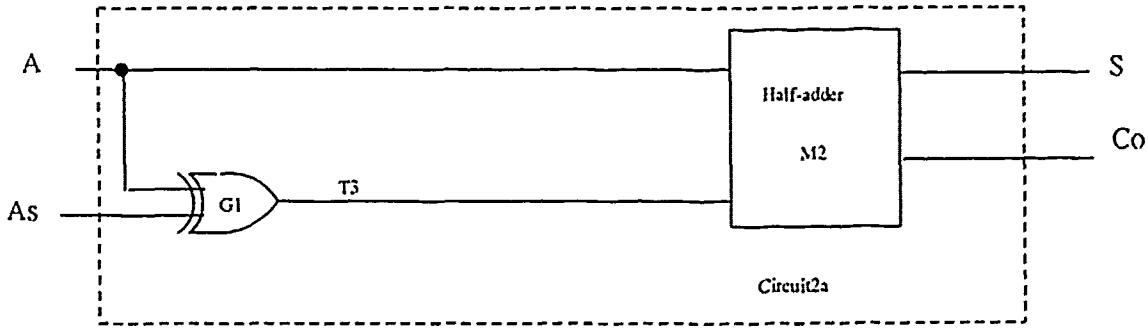


Figure C.4. Circuit schematic for test 2 after specialization.

C.2.3 Test 3 Code and Results

```
/*************************************************************/
/**                    test3 results                        **/
/** Test 3 inputs a circuit where every output is used and  **/
/** no specialization is possible.                          **/
/*************************************************************/

/****************test3 circuit definition*********************/


     test3 :-
       scan(circuit3(a,b,c,as,s,co)).

     circuit3(A,B,C,As,S,Co) :-
       halfadd(B,C,T1,T2),
       halfadd(A,T1,S,Co),
       xor_gate(A,As,T3),
       and_gate(T1,T3,T4),
       or_gate(T2,T4,Co).

/** The following submodule definitions are necessary to    **/
/** run test 3. They are further explained in section C.1.5.**/


     halfadd(A,B,S,C) :-
       xor_gate(A,B,S),
       nand_gate(A,B,T),
       nnot_gate(T,C).

     direction(halfadd(A,B,S,C),[A,B],[S,C]).
     direction(or_gate(A,B,Out),[A,B],[Out]).
     direction(xor_gate(A,B,Out),[A,B],[Out]).
     direction(and_gate(A,B,Out),[A,B],[Out]).
     direction(nnot_gate(A,B),[A],[B]).
```

```
     direction(nand_gate(A,B,C),[A,B],[C]).

     primitive(or_gate).
     primitive(xor_gate).
     primitive(and_gate).
     primitive(nnot_gate).
     primitive(nand_gate).

/************************Test Results************************/

Script V1.0 session started Mon Oct 14 14:47:59 1991

Microsoft(R) MS-DOS(R) Version 4.01
             (C)Copyright Microsoft Corp 1981-1988

A:\THESIS\CODE>Prolog
Bad command or file name

A:\THESIS\CODE>prolog


+-------------------------------------------+
| MS-DOS Prolog-1              Version 2.2  |
| Copyright 1983      Serial number: 0001213 |
| Expert Systems Ltd.                       |
| Oxford U.K.                               |
+-------------------------------------------+

?- [speccode].
speccode consulted.
?- test3.
HeadArgs =[a,b,c,as,s,co]

Body =halfadd(b,c,_102,_103),halfadd(a,_102,s,co),xor_gate(a,as,_118),
and_gate(_102,_118,_125),or_gate(_103,_125,co)
```

```
Goals =[halfadd(b,c,_102,_103),halfadd(a,_102,s,co),xor_gate(a,as,_118),
and_gate(_102,_118,_125),or_gate(_103,_125,co)]

GoalArgsIn =[_103,_125,_102,_118,a,as,a,_102,b,c]
GoalArgsOut =[co,_125,_118,s,co,_102,_103]
 Looking at co
 Looking at _125
 Looking at _118
 Looking at s
 Looking at co
 Looking at _102
 Looking at _103
Unmatched =[]
asserting [T4,T1,T3,T2,a,as,a,T3,b,c]
In analyze 1 halfadd(b,c,T3,T4)
Unmatched args = []
In analyze 1 halfadd(a,T3,s,co)
Unmatched args = []
In analyze 1 xor_gate(a,as,T2)
Unmatched args = []
In analyze 1 and_gate(T3,T2,T1)
Unmatched args = []
In analyze 1 or_gate(T4,T1,co)
Unmatched args = []
Finishing up in analyze

Acc = [or_gate(T4,T1,co),and_gate(T3,T2,T1),xor_gate(a,as,T2),
halfadd(a,T3,s,co),halfadd(b,c,T3,T4)]

ArgsIn = [b,c,a,T3,a,as,T3,T2,T4,T1]
ArgsOut = [T3,T4,s,co,T2,T1,co]
HInArgs = [b,c,a,as]
HOutArgs = [co,s]
NewArgs = [b,c,a,as,co,s]
```

```
NewHead = foo1(b,c,a,as,co,s)

asserting foo1(b,c,a,as,co,s):-[or_gate(T4,T1,co),and_gate(T3,T2,T1),
xor_gate(a,as,T2),halfadd(a,T3,s,co),halfadd(b,c,T3,T4)]

retracting [T4,T1,T3,T2,a,as,a,T3,b,c]
asserting direction(foo1(b,c,a,as,co,s),[b,c,a,as],[co,s]))
Circuit =foo1(b,c,a,as,co,s)
yes
?- halt.

A:\THESIS\CODE>exit
Script completed Mon Oct 14 14:49:06 1991

/**********************End of Test Results******************/
```

Figure C.5 shows the circuit under consideration.

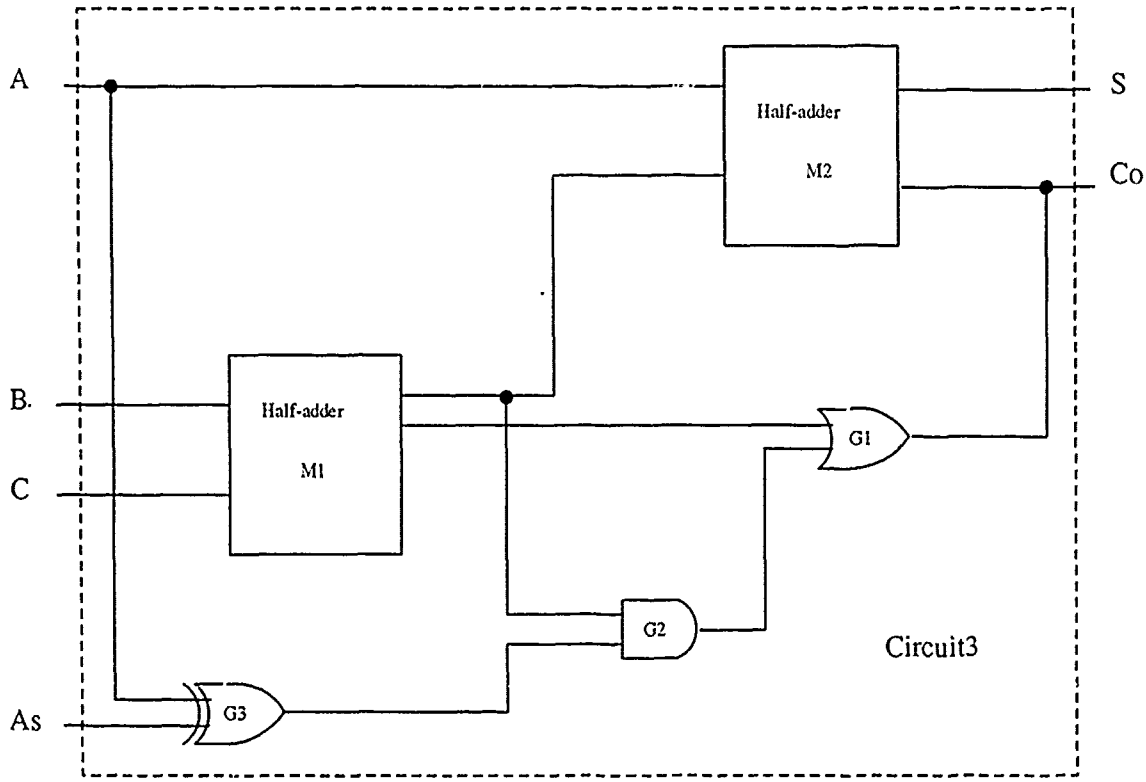


Figure C.5. Circuit schematic for test 3 before and after specialization.

C.2.4 Test 4 Code and Results

```
/*****************************************************************/
/**                      test4 results                          **/
/**  Test 4 is Clocksin's test as described in Chapter 6        **/
/*****************************************************************/

/*****************test4 circuit definition************************/

     test4 :-
       scan(twobit(a1,b1,a2,b2,c,as,s1,s2)).


     twobit(A1,B1,A2,B2,C,As,S1,S2) :-
       addsub(A1,B1,C,As,S1,T),
       addsub(A2,B2,T,As,S2,Unused).


/** The following submodule definitions are necessary to **/
/** carry out test 4. Section C.1.5 further explains the **/
/** following procedures.                                **/


     addsub(A,B,C,As,S,Co) :-
       halfadd(B,C,T1,T2),
       halfadd(A,T1,S,Unused),
       xor_gate(A,As,T3),
       and_gate(T1,T3,T4),
       or_gate(T2,T4,Co).


     halfadd(A,B,S,C) :-
       xor_gate(A,B,S),
       nand_gate(A,B,T),
       nnot_gate(T,C).
```

```
direction(addsub(A,B,C,As,S,Co),[A,B,C,As],[S,Co]).
direction(halfadd(A,B,S,C),[A,B],[S,C]).
direction(or_gate(A,B,Out),[A,B],[Out]).
direction(xor_gate(A,B,Out),[A,B],[Out]).
direction(and_gate(A,B,Out),[A,B],[Out]).
direction(nnot_gate(A,B),[A],[B]).
direction(nand_gate(A,B,C),[A,B],[C]).

primitive(or_gate).
primitive(xor_gate).
primitive(and_gate).
primitive(nnot_gate).
primitive(nand_gate).

/*********************Test Results*********************/

Script V1.0 session started Mon Oct 14 15:04:07 1991

Microsoft(R) MS-DOS(R) Version 4.01
            (C)Copyright Microsoft Corp 1981-1988

A:\THESIS\CODE>prolog


+-------------------------------------------+
| MS-DOS Prolog-1             Version 2.2   |
| Copyright 1983      Serial number: 0001213  |
| Expert Systems Ltd.                       |
| Oxford U.K.                               |
+-------------------------------------------+

?- [speccode].
speccode consulted
```

```
?- test4.
HeadArgs =[a1,b1,a2,b2,c,as,s1,s2]
Body =addsub(a1,b1,c,as,s1,_121),addsub(a2,b2,_121,as,s2,_131)
Goals =[addsub(a1,b1,c,as,s1,_121),addsub(a2,b2,_121,as,s2,_131)]
GoalArgsIn =[a2,b2,_121,as,a1,b1,c,as]
GoalArgsOut =[s2,_131,s1,_121]
 Looking at s2
 Looking at _131
 Looking at s1
 Looking at _121
Unmatched =[T1]
asserting [a2,b2,T2,as,a1,b1,c,as]
In analyze 1 addsub(a1,b1,c,as,s1,T2)
Unmatched args = [T1]
In analyze 1 addsub(a2,b2,T2,as,s2,T1)
Unmatched args = [T1]
In analyze 2 addsub(a2,b2,T2,as,s2,T1)
Unmatched args = [T1]
In specialize 1 addsub(a2,b2,T2,as,s2,T1)
In specialize 2 addsub(a2,b2,T2,as,s2,T1)
Pred = addsub

Goals = [l  :add(b2,T2,_359,_360),halfadd(a2,_359,s2,_368),
xor_gate(a2 as,_375),and_gate(_359,_375,_382),or_gate(_360,_382,T1)]

 Looking at T1
 Looking at _382
 Looking at _375
 Looking at s2
 Looking at _368
 Looking at _359
 Looking at _360
Unmatched = [T5]
looking at [T1]
asserting the following Inargs [T7,T3,T6,T4,a2,as,a2,T6,b2,T2]
```

```
retracting [T1]
asserting [T5,T1]
In analyze 1 halfadd(b2,T2,T6,T7)
Unmatched args = [T5,T1]
In analyze 1 halfadd(a2,T6,s2,T5)
Unmatched args = [T5,T1]
In analyze 2 halfadd(a2,T6,s2,T5)
Unmatched args = [T5,T1]
In specialize 1 halfadd(a2,T6,s2,T5)
In specialize 2 halfadd(a2,T6,s2,T5)
Pred = halfadd
Goals = [xor_gate(a2,T6,s2),nand_gate(a2,T6,_801),nnot_gate(_801,T5)]
 Looking at T5
 Looking at _801
 Looking at s2
Unmatched = []
looking at [T5,T1]
asserting the following Inargs [T8,a2,T6,a2,T6]
retracting [T5,T1]
asserting [T1,T5]
In analyze 1 xor_gate(a2,T6,s2)
Unmatched args = [T1,T5]
In analyze 1 nand_gate(a2,T6,T8)
Unmatched args = [T1,T5]
In analyze 1 nnot_gate(T8,T5)
Unmatched args = [T1,T5]
In analyze 2 nnot_gate(T8,T5)
Unmatched args = [T1,T5]
In specialize 1 nnot_gate(T8,T5)
Unmatched args =[T1,T5]
In check_args1
X = T8
Rest = []
stored inargs = [T8,a2,T6,a2,T6]
deleting T8
```

```
from [T8,a2,T6,a2,T6]
Temp1 = [a2,T6,a2,T6]
in check_args 2
adding T8
to Acc []
checking inargs [T8,a2,T6,a2,T6]
retracting [T1,T5]
asserting all Unmatched args [T5,T1,T8]
Rest = []
Ans = []
Temp1 = []
Temp2 = [nand_gate(a2,T6,T8),xor_gate(a2,T6,s2)]
In analyze 1 nand_gate(a2,T6,T8)
Unmatched args = [T5,T1,T8]
In analyze 2 nand_gate(a2,T6,T8)
Unmatched args = [T5,T1,T8]
In specialize 1 nand_gate(a2,T6,T8)
Unmatched args =[T5,T1,T8]
In check_args1
X = a2
Rest = [T6]
stored inargs = [T8,a2,T6,a2,T6]
deleting a2
from [T8,a2,T6,a2,T6]
Temp1 = [T8,T6,a2,T6]
checking membership of a2
In [T8,T6,a2,T6]
retracting [T8,a2,T6,a2,T6]
asserting [T8,T6,a2,T6]
In check_args1
X = T6
Rest = []
stored inargs = [T8,T6,a2,T6]
deleting T6
from [T8,T6,a2,T6]
```

```
Temp1 = [T8,a2,T6]
checking membership of T6
In [T8,a2,T6]
retracting [T8,T6,a2,T6]
asserting [T8,a2,T6]
checking inargs [T8,a2,T6]
retracting [T5,T1,T8]
asserting all Unmatched args [T8,T1,T5]
Rest = [xor_gate(a2,T6,s2)]
Ans = []
Temp1 = [xor_gate(a2,T6,s2)]
Temp2 = [xor_gate(a2,T6,s2)]
In analyze 1 xor_gate(a2,T6,s2)
Unmatched args = [T8,T1,T5]
Finishing up in analyze
Acc = [xor_gate(a2,T6,s2)]
ArgsIn = [a2,T6]
ArgsOut = [s2]
HInArgs = [a2,T6]
HOutArgs = [s2]
NewArgs = [a2,T6,s2]
NewHead = foo1(a2,T6,s2)
asserting foo1(a2,T6,s2):-[xor_gate(a2,T6,s2)]
retracting [T8,a2,T6]
asserting direction(foo1(a2,T6,s2),[a2,T6],[s2]))
Ans2 = foo1(a2,T6,s2)
Rest = [xor_gate(a2,as,T4),and_gate(T6,T4,T3),or_gate(T7,T3,T1)]
Ans = [foo1(a2,T6,s2)]
Temp1 = [xor_gate(a2,as,T4),and_gate(T6,T4,T3),or_gate(T7,T3,T1),
foo1(a2,T6,s2)]

Temp2 = [halfadd(b2,T2,T6,T7),xor_gate(a2,as,T4),and_gate(T6,T4,T3),
or_gate(T7,T3,T1),foo1(a2,T6,s2)]

In analyze 1 halfadd(b2,T2,T6,T7)
```

```
Unmatched args = [T8,T1,T5]
In analyze 1 xor_gate(a2,as,T4)
Unmatched args = [T8,T1,T5]
In analyze 1 and_gate(T6,T4,T3)
Unmatched args = [T8,T1,T5]
In analyze 1 or_gate(T7,T3,T1)
Unmatched args = [T8,T1,T5]
In analyze 2 or_gate(T7,T3,T1)
Unmatched args = [T8,T1,T5]
In specialize 1 or_gate(T7,T3,T1)
Unmatched args =[T8,T1,T5]
In check_args1
X = T7
Rest = [T3]
stored inargs = [T7,T3,T6,T4,a2,as,a2,T6,b2,T2]
deleting T7
from [T7,T3,T6,T4,a2,as,a2,T6,b2,T2]
Temp1 = [T3,T6,T4,a2,as,a2,T6,b2,T2]
in check_args 2
adding T7
to Acc []
In check_args1
X = T3
Rest = []
stored inargs = [T7,T3,T6,T4,a2,as,a2,T6,b2,T2]
deleting T3
from [T7,T3,T6,T4,a2,as,a2,T6,b2,T2]
Temp1 = [T7,T6,T4,a2,as,a2,T6,b2,T2]
in check_args 2
adding T3
to Acc [T7]
checking inargs [T7,T3,T6,T4,a2,as,a2,T6,b2,T2]
retracting [T8,T1,T5]
asserting all Unmatched args [T5,T1,T8,T7,T3]
Rest = [foo1(a2,T6,s2)]
```

```
Ans = []
Temp1 = [foo1(a2,T6,s2)]

Temp2 = [and_gate(T6,T4,T3),xor_gate(a2,as,T4),halfadd(b2,T2,T6,T7),
foo1(a2,T6,s2)]

In analyze 1 and_gate(T6,T4,T3)
Unmatched args = [T5,T1,T8,T7,T3]
In analyze 2 and_gate(T6,T4,T3)
Unmatched args = [T5,T1,T8,T7,T3]
In specialize 1 and_gate(T6,T4,T3)
Unmatched args =[T5,T1,T8,T7,T3]
In check_args1
X = T6
Rest = [T4]
stored inargs = [T7,T3,T6,T4,a2,as,a2,T6,b2,T2]
deleting T6
from [T7,T3,T6,T4,a2,as,a2,T6,b2,T2]
Temp1 = [T7,T3,T4,a2,as,a2,T6,b2,T2]
checking membership of T6
In [T7,T3,T4,a2,as,a2,T6,b2,T2]
retracting [T7,T3,T6,T4,a2,as,a2,T6,b2,T2]
asserting [T7,T3,T4,a2,as,a2,T6,b2,T2]
In check_args1
X = T4
Rest = []
stored inargs = [T7,T3,T4,a2,as,a2,T6,b2,T2]
deleting T4
from [T7,T3,T4,a2,as,a2,T6,b2,T2]
Temp1 = [T7,T3,a2,as,a2,T6,b2,T2]
in check_args 2
adding T4
to Acc []
checking inargs [T7,T3,T4,a2,as,a2,T6,b2,T2]
retracting [T5,T1,T8,T7,T3]
```

```
asserting all Unmatched args [T3,T7,T8.T1,T5,T4]
Rest = [xor_gate(a2,as,T4),halfadd(b2,T2,T6,T7),foo1(a2,T6,s2)]
Ans = []
Temp1 = [xor_gate(a2,as,T4),halfadd(b2,T2,T6,T7),foo1(a2,T6,s2)]
Temp2 = [xor_gate(a2,as,T4),halfadd(b2,T2,T6,T7),foo1(a2,T6,s2)]
In analyze 1 xor_gate(a2,as,T4)
Unmatched args = [T3,T7,T8,T1,T5,T4]
In analyze 2 xor_gate(a2,as,T4)
Unmatched args = [T3,T7,T8,T1,T5,T4]
In specialize 1 xor_gate(a2,as,T4)
Unmatched args =[T3,T7,T8 T1,T5,T4]
In check_args1
X = a2
Rest = [as]
stored inargs = [T7,T3,T4,a2,as,a2,T6,b2,T2]
deleting a2
from [T7,T3,T4,a2,as,a2,T6,b2,T2]
Temp1 = [T7,T3,T4,as,a2,T6,b2,T2]
checking membership of a2
In [T7,T3,T4,as,a2,T6,b2,T2]
retracting [T7,T3,T4,a2,as,a2,T6,b2,T2]
asserting [T7,T3,T4,as,a2,T6,b2,T2]
In check_args1
X = as
Rest = []
stored inargs = [T7,T3,T4,as,a2,T6,b2,T2]
deleting as
from [T7,T3,T4,as,a2,T6,b2,T2]
Temp1 = [T7,T3,T4,a2,T6,b2,T2]
in check_args 2
adding as
to Acc []
checking inargs [T7,T3,T4,as,a2,T6,b2,T2]
retracting [T3,T7,T8,T1,T5,T4]
asserting all Unmatched args [T4,T5,T1,T8,T7,T3,as]
```

```
Rest = [halfadd(b2,T2,T6,T7),foo1(a2,T6,s2)]
Ans = []
Temp1 = [halfadd(b2,T2,T6,T7),foo1(a2,T6,s2)]
Temp2 = [halfadd(b2,T2,T6,T7),foo1(a2,T6,s2)]
In analyze 1 halfadd(b2,T2,T6,T7)
Unmatched args = [T4,T5,T1,T8,T7,T3,as]
In analyze 2 halfadd(b2,T2,T6,T7)
Unmatched args = [T4,T5,T1,T8,T7,T3,as]
In specialize 1 halfadd(b2,T2,T6,T7)
In specialize 2 halfadd(b2,T2,T6,T7)
Pred = halfadd
Goals = [xor_gate(b2,T2,T6),nand_gate(b2,T2,_2022),nnot_gate(_2022,T7)]
 Looking at T7
 Looking at _2022
 Looking at T6
Unmatched = []
looking at [T4,T5,T1,T8,T7,T3,as]
asserting the following Inargs [T9,b2,T2,b2,T2]
retracting [T4,T5,T1,T8,T7,T3,as]
asserting [as,T3,T7,T8,T1,T5,T4]
In analyze 1 xor_gate(b2,T2,T6)
Unmatched args = [as,T3,T7,T8,T1,T5,T4]
In analyze 1 nand_gate(b2,T2,T9)
Unmatched args = [as,T3,T7,T8,T1,T5,T4]
In analyze 1 nnot_gate(T9,T7)
Unmatched args = [as,T3,T7,T8,T1,T5,T4]
In analyze 2 nnot_gate(T9,T7)
Unmatched args = [as,T3,T7,T8,T1,T5,T4]
In specialize 1 nnot_gate(T9,T7)
Unmatched args =[as,T3,T7,T8,T1,T5,T4]
In check_args1
X = T9
Rest = []
stored inargs = [T9,b2,T2,b2,T2]
deleting T9
```

```
from [T9,b2,T2,b2,T2]
Temp1 = [b2,T2,b2,T2]
in check_args 2
adding T9
to Acc []
checking inargs [T9,b2,T2,b2,T2]
retracting [as,T3,T7,T8,T1,T5,T4]
asserting all Unmatched args [T4,T5,T1,T8,T7,T3,as,T9]
Rest = []
Ans = []
Temp1 = []
Temp2 = [nand_gate(b2,T2,T9),xor_gate(b2,T2,T6)]
In analyze 1 nand_gate(b2,T2,T9)
Unmatched args = [T4,T5,T1,T8,T7,T3,as,T9]
In analyze 2 nand_gate(b2,T2,T9)
Unmatched args = [T4,T5,T1,T8,T7,T3,as,T9]
In specialize 1 nand_gate(b2,T2,T9)
Unmatched args =[T4,T5,T1,T8,T7,T3,as,T9]
In check_args1
X = b2
Rest = [T2]
stored inargs = [T9,b2,T2,b2,T2]
deleting b2
from [T9,b2,T2,b2,T2]
Temp1 = [T9,T2,b2,T2]
checking membership of b2
In [T9,T2,b2,T2]
retracting [T9,b2,T2,b2,T2]
asserting [T9,T2,b2,T2]
In check_args1
X = T2
Rest = []
stored inargs = [T9,T2,b2,T2]
deleting T2
from [T9,T2,b2,T2]
```

```
Temp1 = [T9,b2,T2]
checking membership of T2
In [T9,b2,T2]
retracting [T9,T2,b2,T2]
asserting [T9,b2,T2]
checking inargs [T9,b2,T2]
retracting [T4,T5,T1,T8,T7,T3,as,T9]
asserting all Unmatched args [T9,as,T3,T7,T8,T1,T5,T4]
Rest = [xor_gate(b2,T2,T6)]
Ans = []
Temp1 = [xor_gate(b2,T2,T6)]
Temp2 = [xor_gate(b2,T2,T6)]
In analyze 1 xor_gate(b2,T2,T6)
Unmatched args = [T9,as,T3,T7,T8,T1,T5,T4]
Finishing up in analyze
Acc = [xor_gate(b2,T2,T6)]
ArgsIn = [b2,T2]
ArgsOut = [T6]
HInArgs = [b2,T2]
HOutArgs = [T6]
NewArgs = [b2,T2,T6]
NewHead = foo2(b2,T2,T6)
asserting foo2(b2,T2,T6):-[xor_gate(b2,T2,T6)]
retracting [T9,b2,T2]
asserting direction(foo2(b2,T2,T6),[b2,T2],[T6]))
Ans2 = foo2(b2,T2,T6)
Rest = [foo1(a2,T6,s2)]
Ans = [foo2(b2,T2,T6)]
Temp1 = [foo1(a2,T6,s2),foo2(b2,T2,T6)]
Temp2 = [foo1(a2,T6,s2),foo2(b2,T2,T6)]
In analyze 1 foo1(a2,T6,s2)
Unmatched args = [T9,as,T3,T7,T8,T1,T5,T4]
In analyze 1 foo2(b2,T2,T6)
Unmatched args = [T9,as,T3,T7,T8,T1,T5,T4]
Finishing up in analyze
```

```
Acc = [foo2(b2,T2,T6),foo1(a2,T6,s2)]
ArgsIn = [a2,T6,b2,T2]
ArgsOut = [s2,T6]
HInArgs = [a2,b2,T2]
HOutArgs = [s2]
NewArgs = [a2,b2,T2,s2]
NewHead = foo3(a2,b2,T2,s2)
asserting foo3(a2,b2,T2,s2):-[foo2(b2,T2,T6),foo1(a2,T6,s2)]
retracting [T7,T3,T4,as,a2,T6,b2,T2]
asserting direction(foo3(a2,b2,T2,s2),[a2,b2,T2],[s2]))
Ans2 = foo3(a2,b2,T2,s2)
Rest = []
Ans = [foo3(a2,b2,T2,s2)]
Temp1 = [foo3(a2,b2,T2,s2)]
Temp2 = [addsub(a1,b1,c,as,s1,T2),foo3(a2,b2,T2,s2)]
In analyze 1 addsub(a1,b1,c,as,s1,T2)
Unmatched args = [T9,as,T3,T7,T8,T1,T5,T4]
In analyze 1 foo3(a2,b2,T2,s2)
Unmatched args = [T9,as,T3,T7,T8,T1,T5,T4]
Finishing up in analyze
Acc = [foo3(a2,b2,T2,s2),addsub(a1,b1,c,as,s1,T2)]
ArgsIn = [a1,b1,c,as,a2,b2,T2]
ArgsOut = [s1,T2,s2]
HInArgs = [a1,b1,c,as,a2,b2]
HOutArgs = [s2,s1]
NewArgs = [a1,b1,c,as,a2,b2,s2,s1]
NewHead = foo4(a1,b1,c,as,a2,b2,s2,s1)

asserting foo4(a1,b1,c,as,a2,b2,s2,s1):-[foo3(a2,b2,T2,s2),
addsub(a1,b1,c,as,s1,T2)]

retracting [a2,b2,T2,as,a1,b1,c,as]

asserting direction(foo4(a1,b1,c,as,a2,b2,s2,s1),
[a1,b1,c,as,a2,b2],[s2,s1]))
```

```
Circuit =foo4(a1,b1,c,as,a2,b2,s2,s1)
yes
?- listing(foo3).
foo3(a2,b2,'T2',s2) :-
        [foo2(b2,'T2','T6'),foo1(a2,'T6',s2)] .


yes
?- listing(foo2).
foo2(b2,'T2','T6') :-
        [xor_gate(b2,'T2','T6')] .


yes
?- listing(foo1).
foo1(a2,'T6',s2) :-
        [xor_gate(a2,'T6',s2)] .


yes
?- halt.

A:\THESIS\CODE>exit
Script completed Mon Oct 14 15:05:44 1991

/*********************End of Test Results*********************/
```

The before and after circuit diagrams are shown in Chapter 6. Note that the built in predicate `listing(X)` was used to show the composition of modules which have been specialized.

Bibliography

1. Beard, Nick and Chris Hogger. "A Logical Way to Program," *Systems International*, *17*(4):49–51 (Apr. 1989).
2. Bharath, Ramachandran. "Logic Programming: An Introduction," *Access (USA)*, *6*(4):11–13 (July-Aug 1987).
3. Bover, David C. and Patty E. Brayton. "Sorting Techniques in Logic Programming." *1987 IEEE Region 5 Conference: Electrical Engineering, Challanges in the 1990's, Tulsa, OK, USA, 9-11 Mar. 1987*. 84–88. New York, NY, USA: IEEE Press, 1987.
4. Bratko, Ivan. *PROLOG Programming for Artificial Intelligence* (Second Edition). Reading, MA: Addison-Wesley, 1990.
5. Brezocnik, Z., B. Horvat and M. Gerkes. "Tool for System Design Verification." *System Design: Concepts, Methods, and Tools*. 100–107. Washington, DC, USA: Computer Society of the IEEE, Apr. 1988.
6. Broberg, Harold L. "PROLOG: Programming Language of the Future." *Proceedings - 1987 Frontiers in Education Conference. Seventeenth Annual Conference, Terre Haute, IN, USA, 24-28 Oct. 1987*. 399–403. New York, NY, USA: IEEE Press, 1987.
7. Brown, Frank M. Class Handout Distributed in CSCE 523, Artificial Intelligence, Oct. 1990.
8. Brown, Frank M. Class Handout Distributed in CSCE 631, Notes on Logic, Jan. 1991.
9. Bush, Willaim R., Patrick C. McGeer and Alvin M. Despain. "Experience with PROLOG as a Hardware Specification Language." *Proceedings of the 1987 Symposium on Logic Programming, San Francisco, CA, USA, 31 Aug - 4 Sept 1987*. 490–497. Washington, DC, USA: IEEE Comput. Soc. Press, 1987.
10. Cabodi, Gianpiero, Paolo Camurati and Paolo Prinetto. "The Use of PROLOG for Executable Specifications and Verification of Easily Testable Designs." *FTCS Digest of Papers. 16th Annual International Symposium on Fault-Tolerant Computing Systems, Vienna, Austria, 1-4 July 1986*. 390–395. Washington, DC, USA: IEEE Computer Society Press, 1986.
11. Chang, Chin-liang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. San Diego CA: Academic Press Inc., 1973.
12. Clocksin, W.F. *Logic Programming and the Specification of Circuits*. Technical Report, University of Cambridge: Computer Laboratory, 1985.
13. Clocksin, W.F. "Automatic Specialisation of Standard Designs," *The Computer Journal*, *29*(6):495–499 (Nov. 1986).
14. Clocksin, W.F. "Logic Programming and Digital Circuit Analysis," *Journal of Logic Programming (USA)*, *4*(1):59–82 (Mar 1986).
15. Clocksin, W.F. "A Prolog Primer," *Byte*, *12*(9):147–158 (Aug. 1987).

16. Clocksin, W.F. and C.S. Mellish. *Programming in Prolog*. New York: Springer-Verlag, 1981.

17. Codognet, Christian, Philippe Codognet and Gilberto File. "Very Intelligent Backtracking Method for Logic Programs." *ESOP86 - European Symposium on Programming, Saarbruecken, West Ger, Mar. 1986*. 315-326. New York, NY, USA: Springer-Verlag, 1986.

18. Cohen, Jacques. "View of the Origins and Development of Prolog," *Communications of the ACM*, *31*(1):26-36 (Jan. 1988).

19. Covington, M. "Logic Programming and Artificial Intelligence," *Electron. Wirel. World (UK)*, *95*(1643):879-892 (Sept 1989).

20. Davis, Ruth E. "Logic Programming and PROLOG: A Tutorial," *IEEE Software (USA)*, *2*(5):53-62 (Sept. 1985).

21. de Geus, Aart J. and William Cohen. "A Rule-Based System for Optimizing Combinational Logic," *IEEE Design & Test (USA)*, *2*(4):22-32 (Aug 1985).

22. de Geus, Aart J. and David J. Gregory. "The Socrates Logic Synthesis and Optimization System." *Design Systems for VLSI Circuits. Proceedings of the NATO Advanced Study Institute, L'Aquila, Italy, 7-18 July 1986*. 473-498. Dordrecht, Netherlands: Martinus Nijhoff, 1987.

23. De Micheli, G. "Symbolic Minimization of Logic Functions." *IEEE International Conference onComputer-Aided Design: ICCAD-85. Digest of Technical Papers, Santa Clara, CA, USA, 18-21 Nov. 1985*. 293-295. Washington, DC, USA: IEEE Computer Society Press, 1985.

24. Dettmer, Roger. "Logic Programming - Or How To Make Computers More Thoughtful," *Electronics and Power*, *33*(2):109-113 (1987).

25. Dukes, Michael A., Frank M. Brown and Joanne DeGroat. "Verification of Layout Descriptions Using GES." *Proceedings of the VHDL Users Group Spring 1991 conference, Cincinnati OH., 8-10 Apr. 1991*. 63-72. Menlo Park: Conference Management Services, 1991.

26. Dukes, Michael A., Frank M. Brown and Joanne E. DeGroat. "A Generalized Extraction System for VLSI." *Proceedings of the Fourth Annual IEEE International ASIC Conference and Exhibit, Rochester, New York, 23-27 Sept. 1991*. 4-8.1 - 4-8.4. Piscataway NJ, USA: IEEE Press, 1991.

27. Gero, J.S. "The Use of PROLOG in Computer-Aided Design." *Computer-Aided Design and Manufacture. State of the Art Report.*. 11-24. Maidenhead, Berks, England: Pergamon Infotech, 1985.

28. Gooley, Markian M. "Efficient Reordering of Prolog Programs," *IEEE Transactions on Knowledge and Data Engineering*, *1*(4):470-482 (Dec. 1989).

29. Gregory, David, Aart de Geus and Tim Moore. "Automating Logic Design with Socrates," *Digital design (USA)*, *16*(11):44-46 (Oct 1986).

30. Gregory, David, Karen Bartlett Aart de Geus and Gary Hachtel. "Socrates: A System for Automatically Synthesizing and Optimizing Combinational Logic." *23rd ACM/IEEE Design Automation Conference. Proceedings 1986, Las Vegas, NV, USA, 29 June-2 July 1986*. 79–85. Washington, DC, USA: IEEE Computer Society Press, 1986.

31. Gullichsen, E. "Heuristic Circuit Simulation Using PROLOG," *Integration, the VLSI Journal (Netherlands)*, *3*(4):283–318 (Dec 1985).

32. Hachtel, G., M. Lightner K. Bartlett D. Bostick R. Jacoby P. Moceyunas C. Morrison X. Du and E. Schwarz. "BOLD: The Boulder Optimal Logic Design System." *Proceedings of the Twenty Second Annual Hawaii International Conference on System Sciences: Architecture Track, Kailua-Kona, Hawaii, USA, Jan. 1989*. 59–73. Washington, DC, USA: IEEE Computer Society Press, 1989.

33. Hogger, Christopher John. *Introduction To Logic Programming*. London, England: Academic Press, 1984.

34. Horstmann, Paul W. and Edward P. Stabler. "Computer Aided Design (CAD) Using Logic Programming." *ACM IEEE 21st Design Automation Conference Proceedings 84, Albuquerque NM, USA, 25-27 June 1984*. 144–151. New York, USA: IEEE Press, 1984.

35. Koster, Alexis and Richard A. Hatch. "PROLOG as Every Businessperson's Programming Language: A Survey of the Language," *Journal of Computer Information Systems (USA)*, *28*(3):1–5 (Spring 1988).

36. Kowalski, Robert. "Algorithm = Logic + Control," *Communications of the ACM*, *22*(7):424–436 (Jul. 1979).

37. Kowalski, Robert A. "Early Years of Logic Programming," *Communications of the ACM*, *31*(1):38–43 (Jan. 1988).

38. Kunen, Kenneth. "Negation in Logic Programming," *Journal of Logic Programming*, *4*(4):289–308 (Dec. 1987).

39. Lee, Tsung D. and L. P. McNamee. "Structure Optimization in Logic Schematic Generation." *1989 International Conference on Computer-Aided Design*. 330–333. Los Alamitos, CA: IEEE Computer Society Press, Nov 1989.

40. Ng, K. W. and W.Y. Ma. "Pitfalls in PROLOG Programming," *SIGPLAN Notices (USA)*, *21*(4):75–79 (April 1985).

41. Perkowski, Marek, David Smith and Robert Krzywiec. "Logic Simulation/Design/Verification Environment in Prolog." *Modeling and Simulation, Proceedings of the Seventeenth Annual Pittsburgh Conference, Pittsburgh, PA, USA, 24-28 Apr.1986*. 945–958. Research Triangle Park, NC, USA: ISA, 1986.

42. Robinson, J.A. "Logic Programming - Past, Present and Future," *New Generation Computing*, *1*:107–124 (1983).

43. Roth, Charles H. jr. *Fundamentals of Logic Design*. St. Paul, MN: West Publishing Company, 1979.

44. Sidhu, Deepinder P. "Logic Programming Applied to Hardware Design Specification and Verification." *Seventeenth Annual Microprogramming Workshop, MICRO 17, New Orleans, LA, USA, 30 Oct.-2 Nov. 1984*. 309–313. Washington, DC, USA: IEEE Computer Society Press, 1984.

45. Svanaes, Dag and Einal J. Aas. "Test Generation Through Logic Programming," *Integration, the VLSI Journal (Netherlands)*, 2:49–67 (Sept 1984).

46. Takayoshi, Yokota, Keisuke Bekki and Nobuhiro Hamada. *A VLSI Design Automation System Using Frames and Logic Programming*. Technical Report, Hitachi City. Ibaraki 39-12 Japan: Hitachi Research Laboratory, 1987.

47. Tearnlund, Sten-Ake. "Logic Programming - From a Logic Point of View." *Proceedings - 1986 Symposium on Logic Programming, Salt Lake City, UT, USA, 22-25 Sept. 1986*. 96–103. Washington, DC, USA: IEEE Computer Society Press, 1986.

48. Turing, A.M. "On Computable Numbers with an Application to the Entscheidungsproblem," *London Mathematical Society Proceedings*, *42*(2) (1936-1937).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1991	Master's Thesis

4. TITLE AND SUBTITLE
LOGIC PROGRAMMING IN DIGITAL CIRCUIT DESIGN

5. FUNDING NUMBERS

6. AUTHOR(S)
Joseph W. Eicher, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCE/ENG/91D-03

9. SPONSORING, MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Electronic Technology Laboratory
Wright Laboratories
Microelectronics Division WL/ELED
Wright-Patterson AFB OH 45433-6543

10. SPONSORING, MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The design of large, complex digital circuitry requires highly skilled engineers. Much of the time spent by these engineers in the design phase involves tasks that are repetitive, tedious, and slow. If these repetitive tasks are automated, the engineer can spend more time managing the design process and produce a better-quality design in less time Logic programming can be used to automate design tasks, even those that require a high degree of skill This thesis investigates several aspects of the digital circuit design process that involve pattern-matching paradigms suitable for encoding in the logic programming language Prolog.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Logic Programming,Modelling,Extraction,Simulation,Specialization	198
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL